Rec'd PCT/PTO 22 DEC 2004
PCT/GB 2003 / 0 0 2 8 2 2

10/ 8556

The Patent Office

INVESTOR IN PEOPLE

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

REC'D 01 AUG 2003
WIPO PCT

PRIORITY DOCUMENT

SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed

Dated 15 July 2003

For Official use only

THE PATENT OFFICE
F
28 JUN 2002
LONDON

01JUL02 E729611-3 D10092
P01/7700 0.00-0215035.7

Your reference **Code Generation (UK)**

0215035.7

28 JUN 2002

The
Patent Office

Request for grant of a Patent

Form 1/77

Patents Act 1977

1 Title of invention

Code Generation Method

2. Applicant's details

[X] First or only applicant

2a If applying as a corporate body: Corporate Name

Critical Blue Ltd

Country

GB

2b If applying as an individual or partnership
Surname

Forenames

2c Address The Scottish Microelectronics Centre
The Kings Buildings
West Mains Road
Edinburgh

UK Postcode EH9 3JF

Country GB

ADP Number 8413775001

☐ 2d	Second applicant (if any) Corporate Name Country	
2e	Surname Forenames	
2f	Address UK Postcode Country ADP Number	
3 Address for service		
	Agent's Name	Origin Limited
	Agent's Address	52 Muswell Hill Road London
	Agent's postcode	N10 3JR
	Agent's ADP Number	C03274 7270457002

4 Reference Number

Code Generation (UK)

5 Claiming an earlier application date

An earlier filing date is claimed:

Yes ☐ No ☒

Number of earlier application or patent number

Filing date

15 (4) (Divisional) ☐ 8(3) ☐ 12(6) ☐ 37(4) ☐

6 Declaration of priority

Country of filing	Priority Application Number	Filing Date

7 Inventorship

The applicant(s) are the sole inventors/joint inventors

Yes ☐ No ☒

8 Checklist

			Continuation sheets	
Claims	0		Description	~~70~~ 85
Abstract	0		Drawings	0

Priority Documents	~~Yes~~/No
Translations of Priority Documents	~~Yes~~/No
Patents Form 7/77	~~Yes~~/No
Patents Form 9/77	~~Yes~~/No
Patents Form 10/77	~~Yes~~/No

9 Request

We request the grant of a patent on the basis of this application

Signed: Origin Limited Date: 28 June 2002

(Origin Limited)

Code Generation Method

1 Problem Statement

Most existing modern architectures have a register file centric execution model. Each operation takes register operands and the result is written back into the register file. Each functional unit in the processor has enough access ports to the register file to ensure that it is able to read and write all the required data values to perform the operation. As discussed, this is highly undesirable from an architectural scalability viewpoint. However, it does mean that the code generator does not have to be concerned with the transport of data values to and from functional units. It only has to perform register allocation and the architecture ensures that there are always sufficient communication resources.

A CriticalBlue processor poses a more serious challenge for a code generator. All communications between functional units must be explicitly programmed. For an operation to be performed the code generator must ensure that all the required operands are available at the functional unit performing the operation on the required clock cycle. It is possible that an operation cannot performed on a particular clock cycle because this cannot be achieved, even if the operands have been calculated and are present elsewhere within the processor.

The code generator cannot assume the bus network is fully connected or symmetrical in any way. It will have been optimised for a particular application. There may be many routes to transfer a particular data item to a particular functional unit operand. The code generator needs to choose the route that will have the least impact on the routing of other data items.

Despite this it is possible that the code generator can become deadlocked. That is, previously scheduled operations have blocked all the possible paths for a particular operand so the operation requiring it cannot be scheduled and the code generation cannot complete. This type of failure cannot occur in the code generation for existing machines. The nearest equivalent is a register spill, where a value has to be written to memory because there are insufficient registers to hold all the required values. A CriticalBlue code generator cannot reliably perform a spill operation because it might not be possible to transport the data items to the memory unit to perform the spill itself.

2 Prior Art

Existing architecture do not have to explicitly program the flow of data through the microarchitecture. Connectivity is provided to always ensure that sufficient register file access bandwidth is available. Code generators can schedule operations sequentially without any look ahead since results can always be written back to the register file. They can then be subsequently accessed for later operations.

Code generation for existing architectures normally occurs in the sequential program order. Instructions are emitted one by one with intermediate values being held in registers. For existing architectures this method cannot deadlock. If all registers become filled with values then spill code can be generated. Instruction schedulers may operate over a basic block, or multiple basic blocks in the case of global scheduling. Register allocation and scheduling are generally performed as separate passes. Thus the scheduling algorithm only has to ensure the definition-use relationships are maintained within the code when shuffling instructions.

The most closely related work is that undertaken in the TTA field. Some initial work has been done in this field to examine the issues of code generator and scheduling for such machines. Completed work in this field has concentrated on TTAs that are fully connected.

3 Summary of Contribution

The CriticalBlue code generator generates a graph representation of the data and control flow within a particular block of code. Code generation and instruction scheduling are effectively combined. Critical path analysis is applied to the graph to determine the most performance critical operations in the graph. The most critical operations are then scheduled first so that they are given the best choices of communication routes in the architecture. This is because delays on these operations will have the most impact on overall code performance.

Operations are not scheduled in sequential order. In fact a depth first traversal, directed by the operation priority, is employed. An operation is only scheduled when all the other operations that make use of its results have already been scheduled. Thus the later operations are actually scheduled first.

When an operation is scheduled it is first necessary to check that transport is available to all operations that use its results. There may be multiple users of a particular result. The code generator tries all the possible routes to transport the data items across the buses in the processor. The operation is placed at the latest cycle that allows all results to be transported to the required destinations. Note that this will actually be earlier in the final schedule then the dependent operation so code is being generated in reverse order. It is at this point that the code generation may fail if it is not possible to transport the results. This situation is discussed below. The most direct routes to transport data are chosen if they are available. If they are already in use due to previously scheduled operations then more indirect routes via intermediate holding registers may have to be used. The transport of the data may thus require additional operations to be scheduled to control the intermediate holding registers. The final result of the scheduling is a cycle by cycle representation of the operations to be performed. This is then converted into a machine code representation.

There are further constraints in addition to those imposed by data transports. There may be a limited number of operations that may be issued on a particular cycle. The code generator needs to examine the operations already scheduled for a particular cycle to determine if a particular operation can be issued. There may also be multiple instances of the same type of functional unit. In this case the code generator must select that is the most appropriate to utilise if there are a number available. This choice takes into account the connectivity of the unit to the required destination units.

4 Architectural Overview

4.1 General Philosophy

One of the key requirements of the architecture is to support scalable parallelism. The basic structure of the micro architecture and the operation of the design tools are all focused on that goal.

Extracting parallelism from highly numeric loop kernels is relatively straightforward. Such loops have regular computation and access patterns that are easy to analyse. The nature of the algorithms also tends to lend itself well to parallel computation. The architecture just

needs to balance the availability of computational resources (such as adders, multipliers) and memory units to ensure the right degree of parallelism can be extracted. Such numeric kernels are common for DSPs. The loops tend to lack any complex control flow. Thus DSPs tend to be highly efficient at regular computation loops but are very poor at handling code with more complicated control flow.

Other than in numeric computation loops, C and C++ code tends to be filled with complicated control flow structures. This is simply because most control code is filled with conditional statements and short loops. Most C++ code is also filled with references to main memory via pointers. The result is a code stream from which it is extremely difficult to extract useful amounts of parallelism. In average RISC code, approximately 30% of all instructions are memory references and a branch is encountered every 5 instructions.

General purpose processors for PCs have to deal with this kind of code and extract parallelism from it in order to achieve competitive performance. The complexity of PC processors has mushroomed in recent years to try and deal with this issue. The control logic for a modern PC processor has literally millions of transistors dedicated to the task of extracting parallelism from the code being executed. The extra hardware needed to actually perform the operations in parallel is tiny in comparison with the logic required to find and control them. The main method utilised is to support dynamic out-of-order and speculative execution. This allows the processor to execute instructions in a different order from that specified by the program. It can also execute those instructions speculatively, before it knows for sure whether they should be executed at all. This allows parallelism to be extracted across branches. The difficult constraint is that the execution must always produce exactly the same answer as would result if the instructions were executed strictly one by one in the original order.

The control and complexity overheads of dynamic out-of-order execution are far too high for a CriticalBlue processor. There is a significant cost overhead due to the area occupied by the control logic, not to mention the cost of designing it. Additionally, such logic is not amenable to the scalability requirements of the CriticalBlue architecture.

A number of recent developments in the area of micro architecture have been focused on VLIW type architectures. There is a "back to basics" movement that seeks to place the burden of extracting parallelism on the compiler. The compiler is able to perform much greater analysis to seek parallelism in the application. It is also considerably simpler to develop than equivalent control logic. This is because the control logic must find the parallelism as the program is running so must itself be highly pipelined and suffers from the physical constraints of circuit design. The compiler performs all of its work up front in software with the luxury of much longer analysis time. For most classes of static parallelism, compiler analysis is very effective.

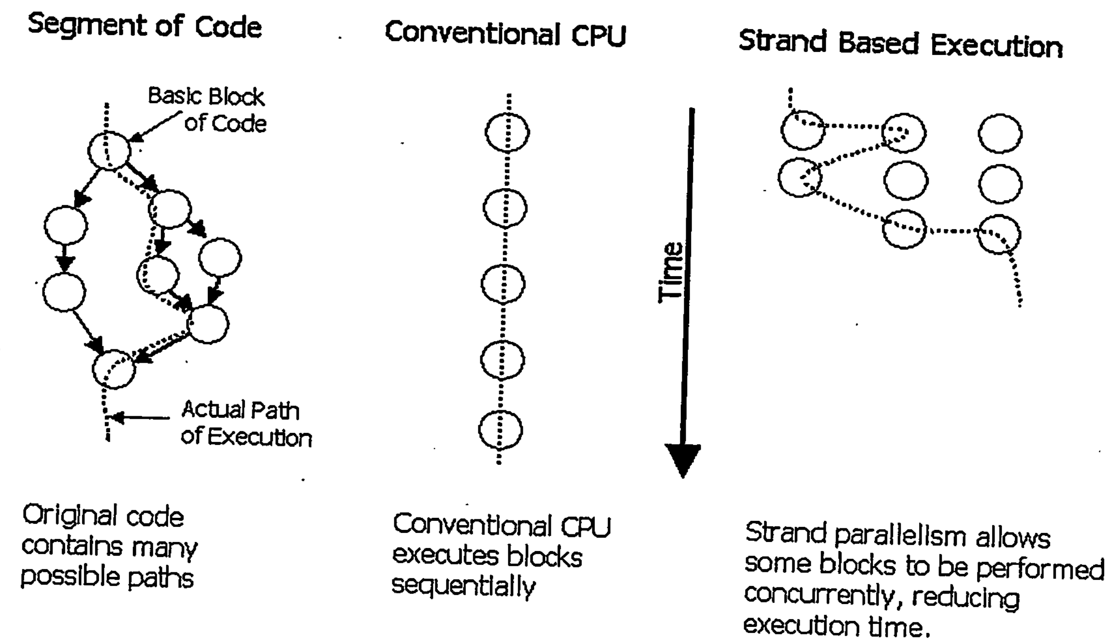
Unfortunately, software analysis is poor at extracting parallelism that can only be determined dynamically. Examples of these are branches and potentially aliased memory accesses. A compiler can know the probability that a particular branch will be taken from profiling information, but it cannot know for sure whether it will be taken on any particular instance. A compiler can also tell from profiling that two memory accesses never seem to access the same memory location, but it cannot prove that will always be the case. Consequently it is not able to move a store operation over a potentially aliased load operation as that might affect the results the program would generated. This restricts the amount of parallelism that can be extracted statically in comparison to that available dynamically.

CriticalBlue employs a unique combination of static and dynamic parallelism extraction. This gives the architecture access to high degrees of parallelism without the overhead of

complex hardware control structures. The software tools perform all the analysis, moving the complexity burden away from the hardware. The CriticalBlue architecture itself is fully static in its execution model. It executes instructions in exactly the order specified by the tools. These instructions may be out of order with respect to the original program, if the tools are able to prove that the re-ordering does not affect the program result. This reordering is called instruction scheduling and is an important optimisation pass for most architectures, and especially for CriticalBlue.

CriticalBlue has a revolutionary execution model that also allows it to perform out-of-order operations that cannot be proved as safe at code generation time. In general it only perform these optimisations if it knows that they will usually be safe at execution time. The hardware is able to detect if the assumptions are wrong and arrange a re-execution of the code that is guaranteed to produce the correct answer in all circumstances. The hardware overhead for this "hazard" detection and re-execution is very small.

The diagram below illustrates the power that this style of execution gives the CriticalBlue architecture:



An example segment of code is shown on the left hand side. This is composed of individual basic blocks. A basic block is a segment of code that is delineated by a branch operation. If execution enters a basic block then all instructions within it will be executed. At the end of the basic block there is a branch instruction that causes execution to continue with one or two different possible successor blocks. The condition upon which the branch is performed is generally calculated in the code of the basic block so it is not possible to know before entering the block which route will be taken. Each execution of the code will produce a particular path of execution through the basic blocks. Certain paths may be considerably more likely than others but any route may be taken.

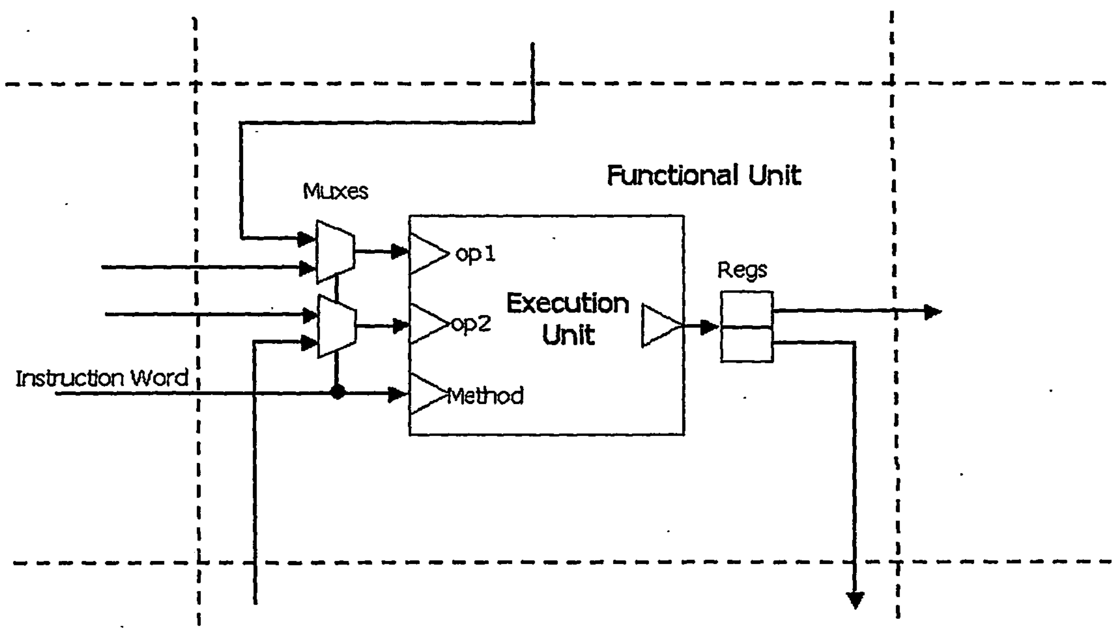
The middle section of the diagram shows the execution in a conventional processor that does not support any kind of out of order execution. This is typical of RISC processor cores. Each basic block as to be executed one after the other, as the branches are resolved.

The right hand section shows the execution model for CriticalBlue. It is able to execute code from a number of different basic blocks in parallel. It does this to increase the amount of parallelism and efficiency of the architecture. It might know that one basic block is very likely to follow another. It can pull instructions forward from the second block to execute in parallel with the instructions of the first. This allows calculations to be started earlier (and thus finish earlier) and to more effectively balance the resource utilisation of the processor. The CriticalBlue scheduling algorithm does this while taking account of the probability that the execution of a particular instruction will be useful.

Executing code speculatively in this manner normally requires a significant hardware overhead. If a particular block should not have been executed then any results it has produced must be discarded. This is referred to as "squashing" the execution. In particular, any store operations that the code has performed must be undone as they could permanently pollute the memory space with incorrect results. CriticalBlue employs mechanisms that allow the benefits of speculative execution while only requiring the minimum of hardware overhead.

4.2 Functional Units

The internal architecture of functional unit is shown below:



The central core of a functional unit is the execution unit itself. It performs the particular operation for the unit. New functional units may be created using user defined execution units. The CriticalBlue tools automatically instantiate the required "glue" blocks around the execution unit in order to form a functional unit. These glue blocks allows the functional unit to connect to other units and to allow the unit to be controlled from the instruction word.

Functional units are placed within a virtual array arrangement. The dashed lines shown on the diagram illustrate this. Individual functional units can only communicate with near neighbours within this array. This spatial layout prevents the architectural synthesis generating excessively long interconnects between units that would significantly impact clock speed.

The control inputs include the segment of the instruction word that controls that particular functional unit. The method selector is passed directly to it. Other fields are used to control the multiplexers that select data from buses. Each operand input to the execution unit may be chosen from one of a number of potential data buses using a multiplexer. In some circumstances the operand may be fixed to a certain bus, removing the requirement for a multiplexer. The number of selectable sources and the choice of particular source buses are under the control of the CriticalBlue architectural optimisation tools.

All results from an execution unit are held in independent output registers. These drive data on point-to-point buses connected to other functional units. The point-to-point nature of these buses minimises power dissipation and propagation delays. Data is passed from one functional unit to another in this manner. The output register holds the same data until a new operation is performed on the functional unit that explicitly overwrites the register.

4.3 Communication Architecture

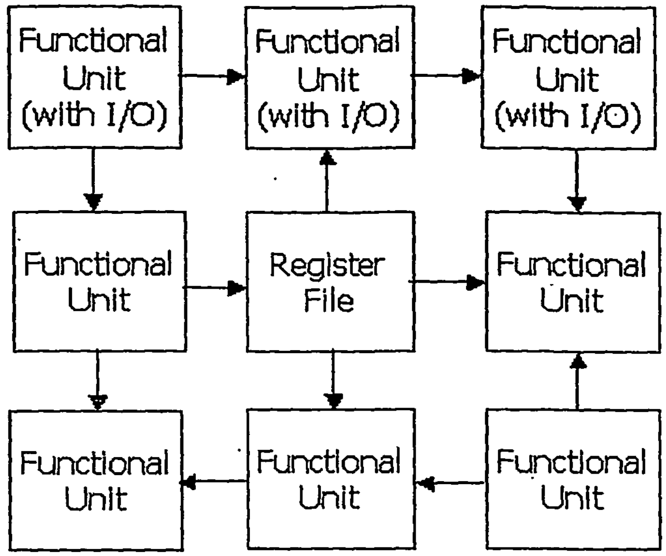
Although the CriticalBlue architecture does have a central register file it is treated like any other implicit functional unit. All accesses to the register file have to be explicitly scheduled as separate operations. Since the register file acts like any other functional unit its bandwidth is limited. The code is constructed so that the majority of data values are communicated directly between functional units without being written to the register file.

Traditional architectures have a centralised register file that has customized access ports to all of the functional units. Access to the register file is implicit in the instruction layout and semantics of the instruction set. The register file is used to feed the operands of the execution units and hold the results generated by them. Unfortunately such a centralised register file imposes a significant restriction on scalability. As the level of parallelism in the instruction stream increases so does the number of access ports required to a centralised register file. These are needed to provide operands to and write back results from all the active execution units. The complexity of the register file grows at approximately N^3 where N is the number of access ports. The register file soon becomes the bottleneck in the design and starts to have a strongly detrimental affect on the maximum clock speed.

Given the requirement to make CriticalBlue highly scalable, communication of all data through a centralised register file is not a viable architectural option. Whenever a functional unit generates a result it is held in an output register until explicitly overwritten by a subsequent operation issued to the unit. During this time the functional unit to which the result is connected may read it.

A single functional unit may have multiple output registers. Each of these is connected to different functional unit or functional unit operand. Since all connection buses are single source/sink there is a one-to-one correspondence between output registers and connections. The output registers that are overwritten by a new result from a functional unit are programmed as part of the instruction word. This allows the functional unit to be utilised even if the value from a particular output register has yet to be used. It would be highly inefficient to leave an entire functional unit idle just to preserve the result latched on its output. In effect each functional unit has a small, dedicated, output register file associated with it to preserve its results.

An example functional unit array is shown in the diagram below:



Given the connectivity limitations of the functional unit array, not every unit is connected to every other. Thus in some circumstances a data item may be generated by one unit and needs to be transported to another unit with which there is no direct connection. The placement of the units and the connections between them is specifically designed to minimise the number of occasions on which this occurs. The interconnection network is optimised for the data flow that is characteristic of the required application code.

To allow the transport of such data items, any functional unit may act as a repeater. That is it may select one of its operands and simply copy it to its output without any modification of the data. Thus a particular value may be transmitted to any operand of a particular unit by using functional units in repeater mode. A number of individual "hops" between functional units may have to be made to reach a particular destination. Moreover, there may be several routes to the same destination. The code generator selects the most appropriate route depending upon other operations being performed in parallel.

There are underlying rules that govern how functional units can be connected together. Local connections are primarily driven by the predominate data flows between the units. Higher level rules ensure that all operands and results in the functional unit array are fully reachable. That is, any result can reach any operand via a path through the array using units as repeaters. These rules ensure that any code sequence involving the functional units can be generated. The performance of the code generated will obviously depend on how well the data flows match the general characteristics of the application. Code that represents a poor match will require much more use of repeating through the array.

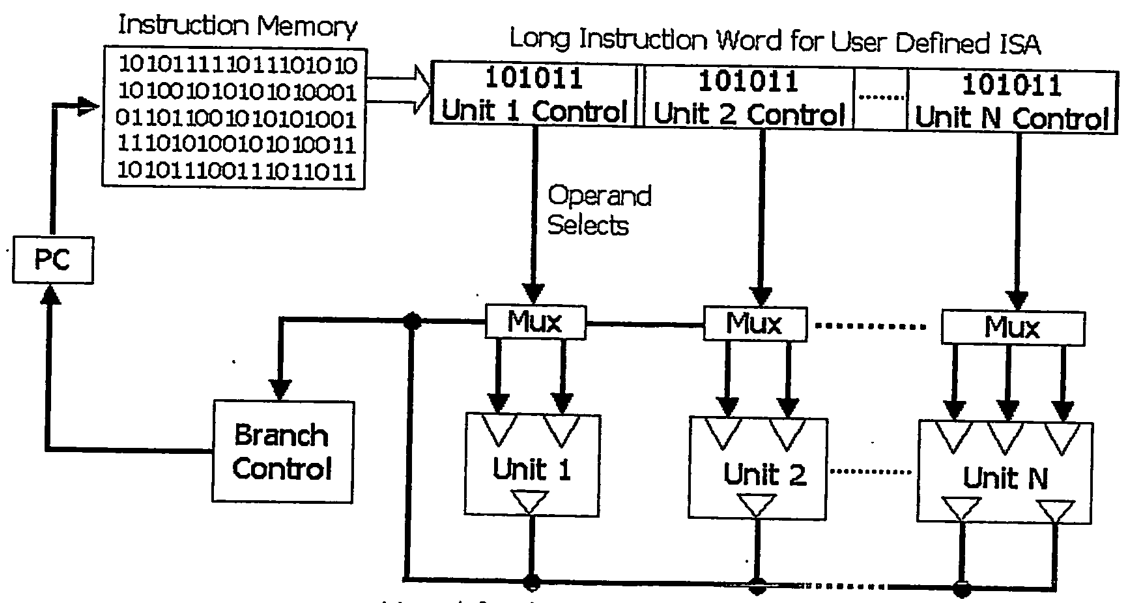
CriticalBlue uses a truly distributed register file concept. The main register file does not form a central bottleneck that would severely hamper the scalability of the architecture. The majority of communication in the CriticalBlue architecture occurs directly between functional units without a requirement to use the central register file. Small blocks of registers local to each functional unit hold data values until they are required by subsequent operations. The connections between the functional units and their placement are optimised synergistically to adapt to the data flow present in the application.

4.4 Instruction Representation

CriticalBlue uses a Very Large Instruction Word (VLIW) format. This enables many parallel operations to be initiated on a single clock cycle, enabling massive parallelism. The actual width is not fixed by the architecture and is under user control. Shorter widths tend to be more efficient in terms of code density but poorer in extracting parallelism from the application.

The instruction format is not fixed either and is dependent upon the execution units the user defines for a particular processor. Unlike many contemporary VLIW architectures, CriticalBlue uses a flat decode structure. This means that a particular execution unit is always controlled from a specific group of bits in the instruction word. This makes the instruction decoding for the architecture very straightforward. Other VLIWs tend to just bundle a number of independent operations into a single instruction word. They still require quite complex decode logic to direct different operations to the appropriate execution units.

The diagram below illustrates the basic instruction decode and control paths of a CriticalBlue processor:



User defined execution units (automatically replicated and connected by tools)

The instruction (or code) memory holds the representation of the operations in the customized format for the processor. A new instruction word is fetched on each clock cycle. Each block of bits in the instruction word is used for controlling a particular execution unit.

The bits in the instruction word are used to control multiplexers that direct data from the interconnection network to the operand inputs of the execution unit. A further field selects the particular method that a unit should execute. Results from the execution units are routed back to the interconnection network to be used by subsequent operations.

A branch control unit allows the architecture to execute new blocks of code by loading a new value in the PC (Program Counter). If a branch is not executed then the PC is just incremented on each cycle to execute code sequentially from the code memory.

The diagram represents a slight simplification of how the architecture actually operates but demonstrates the key features. In particular, the instruction word layout is not completely flat. If it were then the width of the instruction word would grow with the number of execution units in the system, potentially reaching unwieldy widths. The representation would also be highly inefficient as a number of execution units will generally be unused on each cycle, and thus the bits controlling them would be wasted.

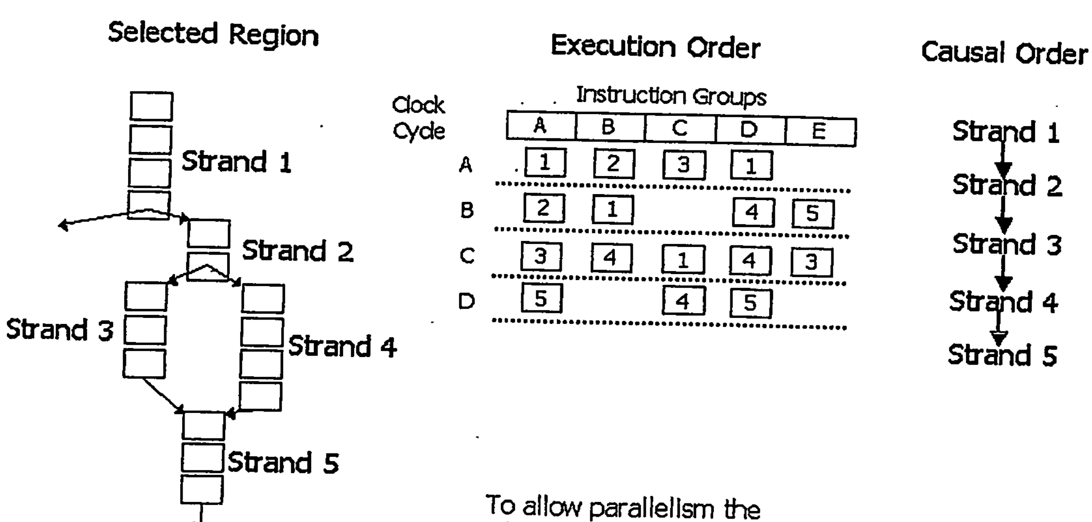
The architecture actually allows a number of execution units to be controlled from the same block of bits in the instruction word. An operation code field selects which of the execution units is selected. The design tools analyse the usage coincidence of different execution units. Units that are rarely used on the same cycle are allocated to the same group of bits in the instruction word, improving code density with minimal impact on performance.

4.5 Strand Execution Model

One of the central innovations of the CriticalBlue architecture is its "strand" based execution mechanism. These are rather like threads but represent a much lower level construct that is present in the architecture to support out-of-order execution.

A strand represents a particular sequential group of operations that is being executed on the machine. Many strands may be executed simultaneously. Each individual operation that is performed belongs to a particular strand. Whenever an instruction word is executed it may contain operations that associated with a number of different strands:

The diagram below illustrates the relationship between strands and basic blocks:



The original basic blocks impose a order on the code. Operations in an strand must not affect an earlier one.

To allow parallelism the schedule mixes up instructions from different strands. Strand numbers are shown in the boxes.

But the correct order of memory accesses must be maintained.

The different colours on the left hand side of the diagram represent different basic blocks in the code. The individual blocks represent the individual operations that are present in each of the basic blocks. The last operation is a branch that determines which basic block will be executed next. Each of the basic blocks is allocated a different strand number.

The middle section of the diagram shows a potential instruction schedule generated for a CriticalBlue processor. Operations from different strands (i.e. basic blocks) may be issued during the same clock cycle. The original strand number is shown in each operation. The order of operations within a particular strand is always maintained. The order of operations between strands does not have to be maintained, allowing much greater scheduling freedom for the architecture. Although the scheduler is able to perform operations out of order between strands it will only do so if that is unlikely to lead to a hazard. The hardware is able to recover from a hazard but there is a performance penalty to doing so.

This mechanism allows instructions to be issued out of order. However, if the correct results are to be produced by the architecture then the data flows between strands that would occur if they were executed in the correct order must be maintained. The right hand part of the diagram shows that the logical order of the strands is always the same. A result generated by an operation in strand 3 should never influence the calculations performed in strand 2. That could never happen if the instructions were executed in order. In effect the architecture has two different time domains. There is the physical time domain of the order of instruction execution. There is also the logical time domain that was present in the original program that must be preserved in order to maintain the original program semantics.

The CriticalBlue tools can determine the correct ordering of most operations statically. The main exception to this is memory operations, where the addresses cannot be determined at compile time.

4.6 Region Based Execution

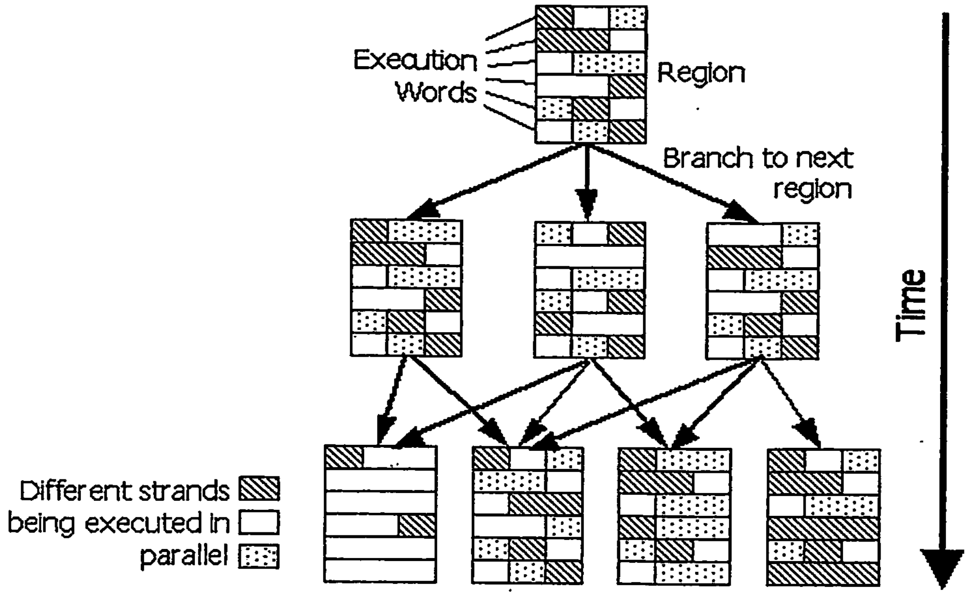
In the CriticalBlue architecture all execution is performed within blocks of code called regions. This simplifies the implementation of both the instruction scheduling and the strand control mechanisms in the hardware.

A region is a block of code that only has a single entry point but potentially many exit points. The analysis performed by the CriticalBlue tools is able to form groups of basic blocks into regions. Regions are often used as the basic arena in which global scheduling optimisations are performed. Global scheduling refers to the movement of instructions across branches as well as within individual basic blocks. Global scheduling is a considerably more difficult problem than basic block scheduling.

In the CriticalBlue architecture, regions are always executed fully. If the region contains a number of internal branches to basic blocks outside of the region then they are not resolved until the end of the region reached. The compiler constructs the regions from basic blocks so that they contain the most likely execution paths through the basic blocks. A region is able to perform a multi-way branch to select one of a number of different successor regions.

All strands are limited to the lifetime of a single region. The architecture is able to execute operations out of order within a particular region. Out of order execution and any resulting hazards are resolved at the end of the region and then execution continues on to another region, which may itself issue operations out of order.

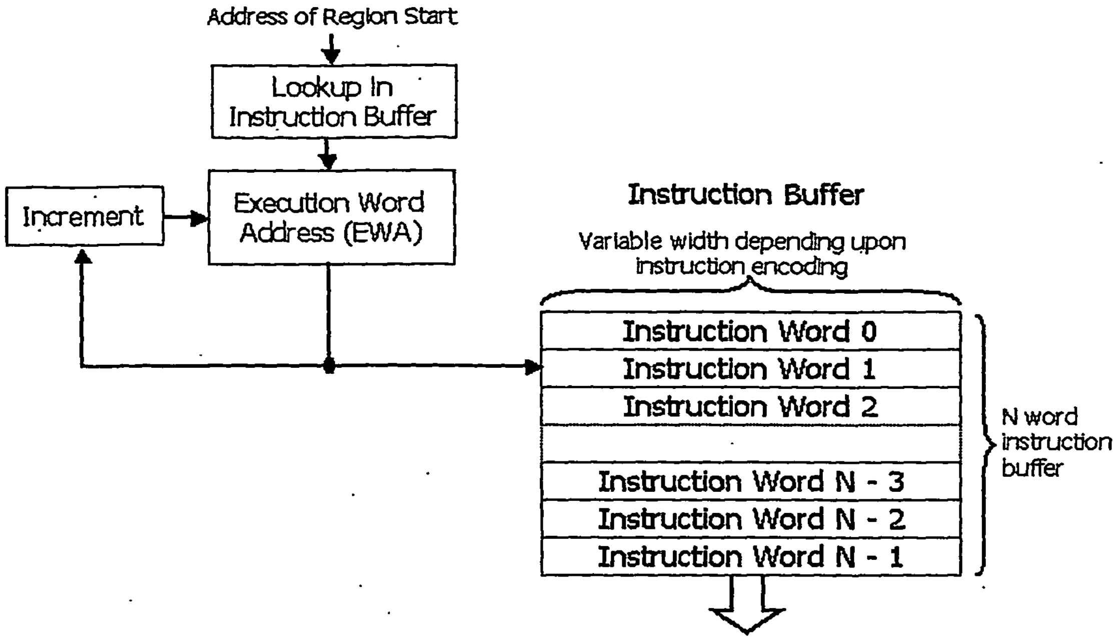
The diagram below illustrates an example set of regions and the relationships between them. It shows the execution of the individual strands within each region:



If a hazard is detected during execution then the sequential semantics of the strands have not been properly preserved. The architecture must be able to recover from this situation with as little overhead as possible.

Upon detecting a hazard in a particular strand the results generated for that and any later (i.e. higher numbered) strands may be incorrect. The architecture allows execution to continue until the end of the region, when the strands will be completed. Any results from the hazard, and any higher, strands are discarded. The architecture then re-executes the code from the start of the region again. Since lower numbered strands have already been successfully completed they are not executed a second time. The architecture includes logic to block operations from those strands. Since the lower strands have completed and generated their results the hazard strand is able to execute correctly, utilizing any required results from the lower strands. If another, even higher numbered, strand generates a hazard then the region may be repeated a second time. When all strands have successfully completed the processor may move onto the successor region.

Of course, the goal of the CriticalBlue architecture is to execute all strands successfully on the first attempt. The compiler does extensive analysis to ensure that the chances of hazards are small. The key is that the compiler doesn't have to prove that a hazard cannot happen. The re-execution mechanism will ensure correct completion of the strands if required. It does this with a minimum of hardware overhead. The size of regions is limited to a few tens of instructions so that the overhead of any re-execution of the region is not too great.

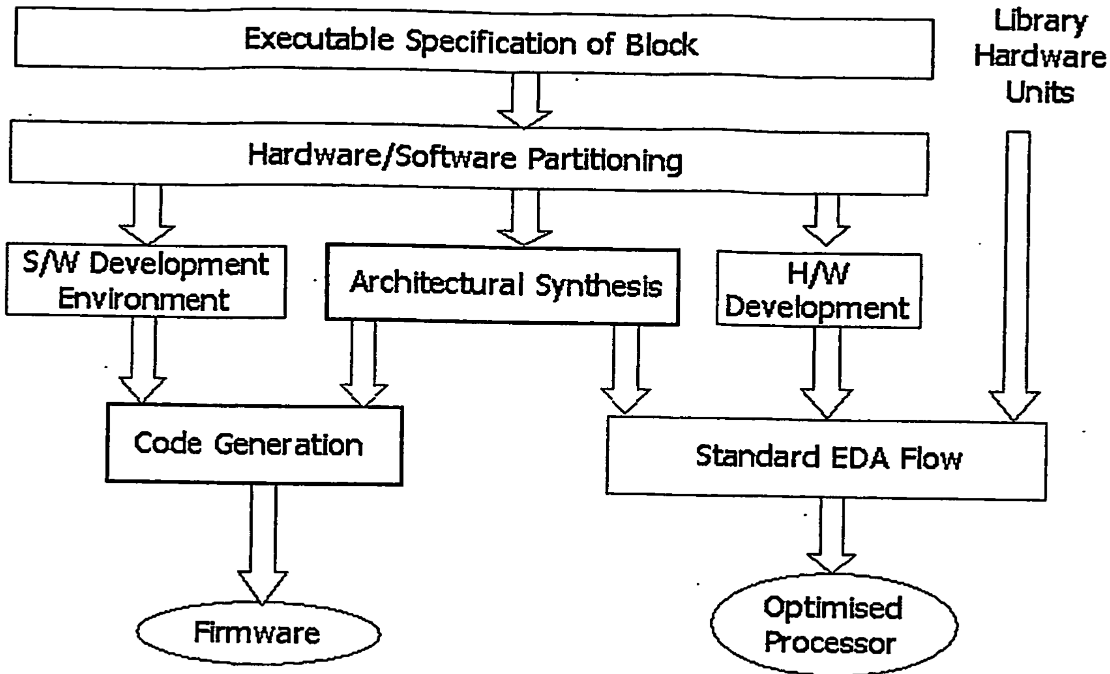


5 Tool Chain

5.1 Tool Flow Overview

The CriticalBlue tools work alongside existing design flows for ASIC or FPGA design. They are used during the front end architectural/system level design phases.

The overall design process is shown below:



As previously discussed, the simulation environment is used to measure perform in order to partition the system between hardware and software. During this process a number of trial processor architectures will be produced. The original C++ code is annotated using the MetaC constructs to show which objects are implemented in hardware.

The partitioned system specified in C++ can be analysed using the tools. This generates an optimised processor that includes all the hardware execution blocks specified in the C++. It effectively forms a processor with a user customised instruction set. The tools can also use the profiling results from the simulation to automatically optimise the connections between the execution units. They automatically replicate individual execution units that have high utilisation in performance critical sections of the application. This allows a number of parallel operations in order to improve performance.

The tools generate a description of the processor at RT-Level (in either Verilog or VHDL). This description can then be directly synthesised along with the individual hardware units into hardware using a standard EDA tool flow. The final target can be an ASIC design or an FPGA.

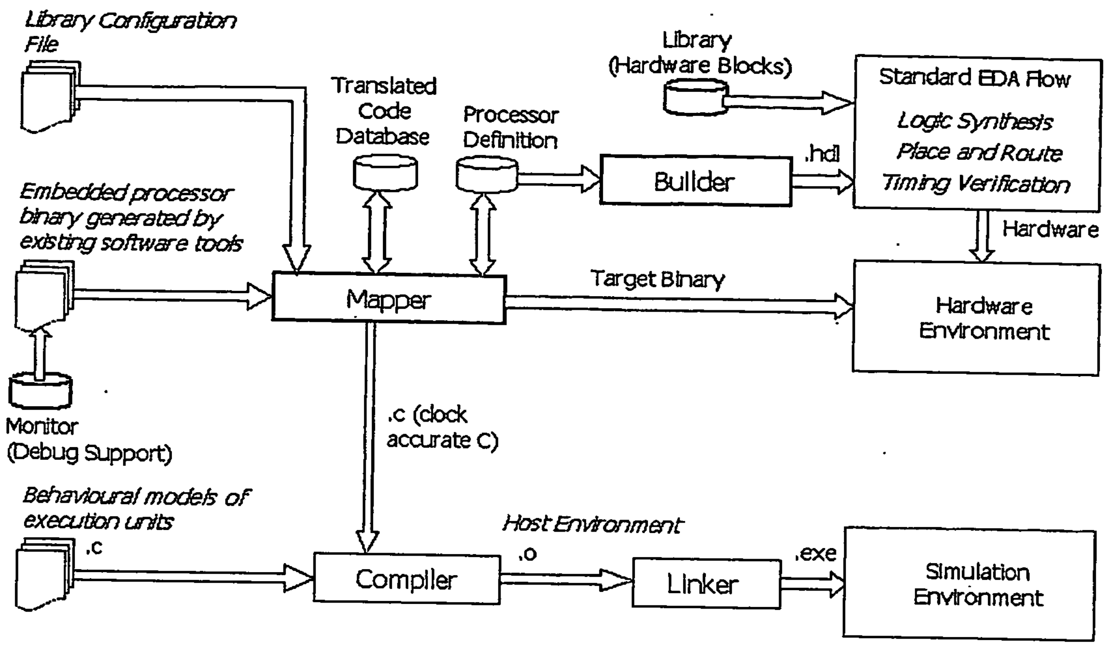
Other tools are able to read the instruction set information generated by the processor optimiser. They can then generate software to run on the customised processor from the C++ code. The whole of the C++ language is supported without restrictions. References to objects marked as being implemented in hardware are automatically converted into instructions to access the appropriate execution unit in a completely seamless manner. The engineer does not have to write any new code to interface between hardware and software, as is traditionally required.

A tool is also provided that converts a C++ description of the behaviour of a hardware block directly into RTL HDL. It can be directly synthesised into hardware. This means that even more of the system can be specified in C++ directly. This synthesis capability will be limited to simple expressions in the first version of the CriticalBlue tools but will be significantly expanded in the second release. At that stage it will support all the constructs in Verilog and VHDL at RT-Level that have direct parallels in C++. This will allow complex execution units to be specified directly in a subset of C++.

5.2 The Individual Tools

The CriticalBlue product consists of a number of individual tools that will be used by engineers. The overall design chain is subdivided into these tools to provide greater flexibility and improve interoperability with systems supplied by other EDA vendors.

The overall tool chain and relationships between the tools is shown below:



Looking from left to right, there are three complete chains for using the tools. One is for designing a CriticalBlue processor, one for generating code to run on it and one for simulation (potentially when no hardware is available). Each of these routes utilise software development tools that generate code for another processor architecture. This is not shown on the diagram. The initially supported environment will be the ARM Development System.

The upper route is for generating a CriticalBlue processor. It passes through the MetaMapper and MetaBuilder. The MetaMapper analyses executable code for another processor architecture, a library configuration file and user performance requirements. The output from MetaMapper is a processor definition file. This describes the optimised connections between units in the hardware library. The MetaBuilder can then be used. It reads that file and generates the connections in a form suitable for use with existing EDA tools. This is combined with the implementation of the individual hardware units from the library. A standard EDA tool flow is used to generate the final hardware.

The middle route generates code for a previously generated CriticalBlue processor. It also passes through the MetaMapper. It directly generates a binary executable that can be used on the CriticalBlue processor. It reads the previously written processor definition. MetaMapper also maintains a database of previously translated blocks of code for the same processor. This avoids having to translate an entire application if only a small portion of the code has changed. This is often the case during development.

Finally, the lower route is used for simulation. The MetaMapper is used to generate cycle and bit accurate code. Rather than outputting a binary for execution on a real processor, the MetaMapper can produce C or C++ code. This can then be compiled with a standard native compiler and combined with the code to implement behavioural models of any

specialised hardware units in the processor. Simulations can be performed for different CriticalBlue processor configurations. This is done using the upper route. New processor definition files are produced, but without using the MetaBuilder to actually generate the hardware form of the processor.

5.3 MetaMapper

This is the main optimisation tool. It reads executable files generated by an existing compiler for a particular processor architecture. Release 1.0 of the CriticalBlue tools will support the ARM architecture. The MetaMapper is able to translate ARM executable files into the correct form for execution on a CriticalBlue processor. It can operate in two distinct modes. In the architectural synthesis mode it is used to construct the architecture of a customised CriticalBlue processor. The tool generates optimised connections between execution units as required by customer performance requirements. In code generation mode it generates code that can run on that particular processor. Both modes are used during the development of a particular chip design. Once the processor has been fixed the code generation mode may be used to modify the software that is to be run on the processor. This is how the technology provides much greater flexibility over fixed hardware. This cannot be modified once the design is completed. The uses a processor definition file that describes the structure of the processor. This is written by architectural synthesis mode and read by the code generation mode.

5.4 MetaBuilder

Reads the processor definition file and generates structural HDL that describes the connections between the hardware execution units in the processor. This can then be input into a standard EDA flow along with the execution unit descriptions to produce the full hardware for a CriticalBlue processing system.

5.5 MetaMonitor

This is a monitor program for supporting debugging of programs running on a CriticalBlue processor. It is linked in as part of the customer program. It communicates with a debugger running on a host computer using a standard remote debugging interface. This allows the user to set breakpoints and single step the program etc. from the host computer. The MetaMonitor uses special code automatically produced by the MetaMapper to emulate an existing chip architecture. Again, this will be ARM for version 1.0 of the CriticalBlue tools. Debugging can proceed as though the software was actually running on an ARM chip.

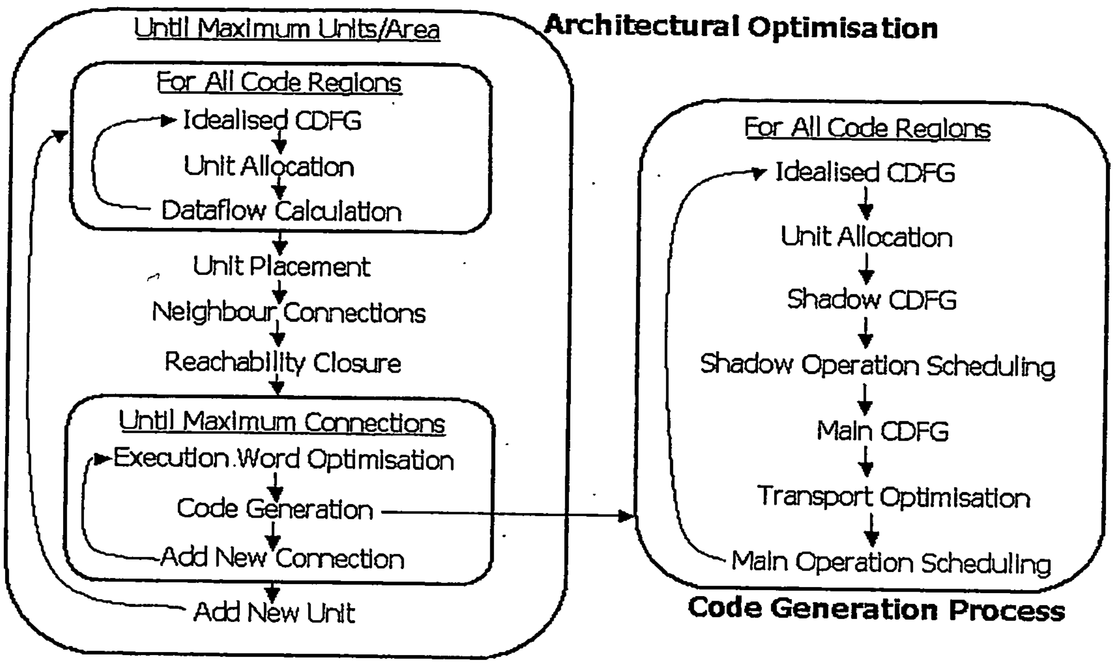
5.6 MetaLibrary

This is a library of hardware blocks that are used to construct CriticalBlue processors. It consists of control blocks for the CriticalBlue processor and standard computational and logical units commonly found in processor designs. The control blocks will be developed by CriticalBlue and be protected by its patents. These control blocks will be used in every CriticalBlue processor. It is from this that CriticalBlue's royalties will be derived. The other units will be obtained from the public domain or from licensing agreements with library vendors. Future library versions may be aimed at particular vertical markets, containing hardware blocks of particular use to certain applications. The customer can extend the MetaLibrary to add hardware blocks for their particular application.

6 Optimisation Flow

6.1 Flow Diagram

The diagram below shows the flow of individual optimisation steps for both architectural optimisation and the code generation process. This document only covers the code generation process.



Once the architecture has been fixed and new code is to be targeted to a processor then only the code generation process needs to be performed. As shown the code generation process is actually a subset of the full architectural optimisation.

6.2 Code Generation Steps

This section details the individual steps that are performed in order to generate code for a particular architecture. This may be a previously generated fixed architecture or it may be a candidate architecture that is being simulated during the architectural optimisation process.

The individual steps are performed on a CDFG that represents a particular region in the code. Thus all the regions that form the code image must be processed in turn. Control flow analysis is performed to identify each of the separate functions and then each of the individual regions within each function.

6.2.1 Idealised CDFG

The first step is to produce an idealised CDFG. This involves creating a CDFG by translating the relevant code from the source binary image. An idealised CDFG does not include many of the explicit register file read and write operations that are required to access items from the register file. An idealised CDFG assumes that data can flow directly from one operation to the next without needing to be written to the register file. This idealised CDFG representation cannot be used for final code generation but allows the predominate data flows in the code to be captured. An idealised CDFG also avoids dependency arcs between potentially aliased memory accesses (although they are still

generated for definitely aliased accesses). Thus the CDFG and data flow is not unnecessarily serialised by the existence of potential memory hazards.

An idealised CDFG is constructed as a first step in order to drive the next stage of unit allocation. To work efficiently the unit allocation needs to know the units from which operands are obtained and to which results are ultimately transported. This information is obfuscated in a non-idealised and unoptimised CDFG as most accesses will be to the register file. By using information about the data flow the unit allocation can make efficient choices about which unit to allocate a particular operation to if there is a choice of multiple units. The underlying assumption is that the majority of optimisations introduced in the CDFG by its idealised creation will ultimately be available by applying subsequent transport optimisations on an unoptimised CDFG.

6.2.2 Unit Allocation

The purpose of the unit allocation is to fix the physical functional unit that will perform each operation in the CDFG. Where there is only a single functional unit of the required type for an operation available this process is obviously trivial. However, in order to exploit parallelism in the code, in many cases there will be a range of functional units of the same type from which to choose. The unit allocation must both balance the usage of all the functional units and make spatially sensible choices so that units are used that are close to the functional units that generate the input operands required and close to the units that will ultimately consume the results. Making such selections minimises the overhead and latency introduced by having to transport data between functional units via copy operations. The unit allocation operates on the idealised CDFG so that accesses to the register file do not hide the true source and destination of particular data items.

6.2.3 Shadow CDFG

The next stage is to generate a shadow CDFG by translating the source instructions. The shadow CDFG is used for generating shadow code that is only executed when debugging is being performed on the code. It allows the machine state that would be generated by the original binary code to be reproduced to on a CriticalBlue processor.

The units allocated for particular operations are obtained form the previous unit allocation phase. Thus the allocated units for the operations is identical between the shadow code and the main code.

6.2.4 Shadow Operation Scheduling

This stage generates the actual microcode representation for the shadow code. The individual operations are scheduled as efficiently as possible.

6.2.5 Main CDFG

During this phase the main CDFG is generated by translating the source instructions. The generated CDFG is identical to that generated for the shadow code except that the additional shadow operations are not included. The partitioning of the code between strands is identical. Moreover, the same unit allocations as originally determined by the unit allocation phase are also employed. Transport allocations are performed in the same manner as described for the shadow CDFG construction.

6.2.6 Transport Optimisation

During this phase the default transports allocated during the main CDFG construction are optimised. The initial usage of default routes for transports results in unnecessary serialisation of particular operations that share elements of their transport routes. This reduces the amount of parallelism available and degrades overall performance.

The purpose of the transport optimisation phase is to improve the transport operations around the nodes in the CDFG in the order of their overall criticality. Thus the more critical operations are given the widest choice of alternative transport routes. The CDFG is rewritten to utilise more direct or efficient transport routes where possible. The transport optimisation phase is also responsible for storing requests for new connections between functional units in the architecture. These connection requests are used during the architectural optimisation to select addition physical connections to be added to the architecture.

6.2.7 Main Operation Scheduling

The main operation scheduling maps the optimised CDFG onto the architecture. This generates the actual microcode for the main code for the application. The CDFG for the main code may also contain weak dependency arcs. A decision is made during the scheduling process about whether those dependencies should be observed or not, in order to generate denser code.

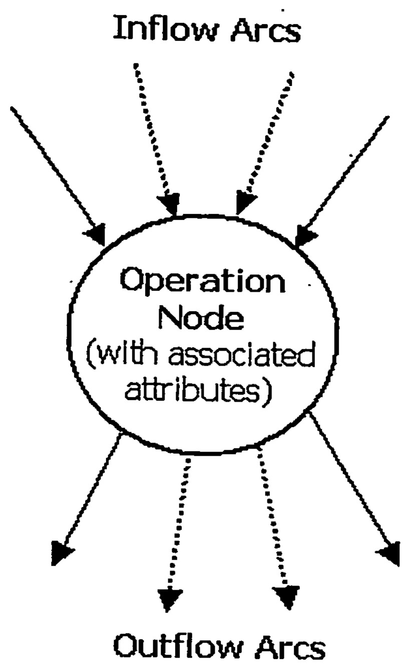
7 Control/Data Flow Graph Representation

7.1 CDFG Representation

The Control and Data Flow Graph (CDFG) is a critical data structure within MetaMapper. It is used to represent both the control and data flow of a sequence of code. The graph is constructed by analyzing source machine code. The graph representation elicits the data flow between operations and their other dependencies. The representation allows the ordering and timing constraints of operations to be shown will avoiding unnecessary restrictions on the ordering of operations.

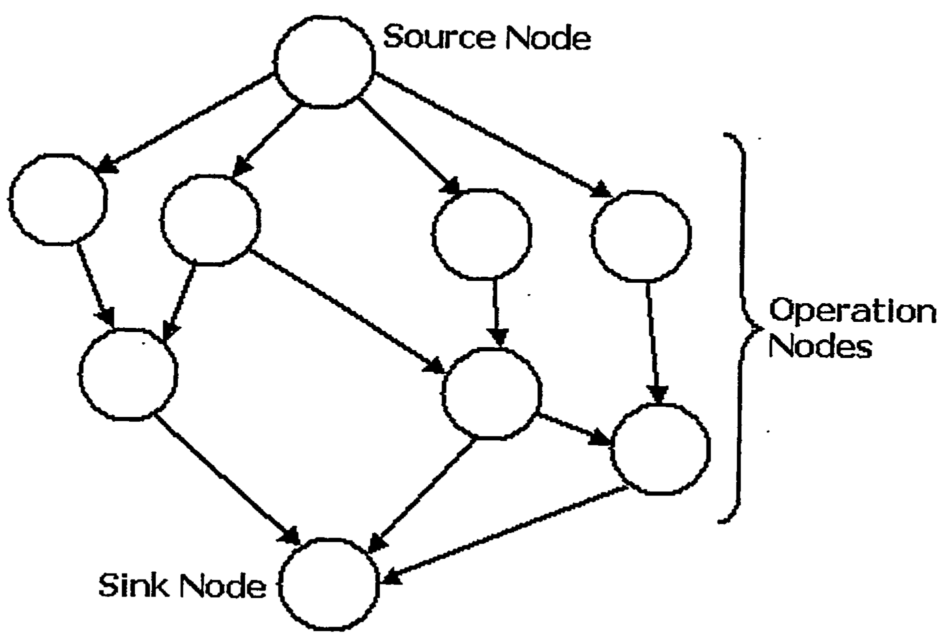
The CDFG is a Directed Acyclic Graph (DAG). A CDFG is constructed for each region being translated. The graph construction must ensure that its acyclic property is maintained, as the scheduler is unable to generate code sequences for cyclic graphs. The nature of code data and control flow is such that this is relatively easy to achieve. Loops in the control flow are not represented within a region itself but by a branch to the start of the region containing the loop. This branch is considered to be external to the region and, as such, does not require a cyclic arc in the graph.

The fundamental component of the CDFG is the node. This is illustrated in the diagram below:



An operation node has a number of associated attributes that describe the operation to be performed. Each node also has a number of inflow and outflow arcs. A node must have at least one inflow arc and one outflow arc. The only exceptions are the source and sink nodes at the start and end of the CDFG, respectively.

The diagram below shows the structure of a typical CDFG:



The first node is the source node for the CDFG. There are various operation nodes that are generated as part of the translation process. There are various dependencies

between those nodes that show the ordering constraints between them. Finally, there is a sink node representing the end of the CDFG.

Operation scheduling is performed from the end of the CDFG (i.e. the sink node) to the source node. A given node cannot be issued in the schedule until all its dependent nodes have been issued. The node can then be issued earlier in the schedule than the earliest of its dependents. This is a depth first traversal of the CDFG.

The following describes the various types of nodes and arcs that may appear in a CDFG:

7.1.1 Node Types

7.1.1.1 Source Node

The source node is the very first node in the CDFG. It has no inflow arcs. It is a virtual node only present to allow easy traversal of the CDFG. It does not result in an operation being generated in the final code sequence.

7.1.1.2 Operation Nodes

Operation nodes are generated as part of the translated process. Each operation node has various attributes associated with the operation that it represents. These are dependent upon the type of operation. However, all operations have an associated unit type and method. These show which particular type of unit will execute the operation and the particular method to be used.

7.1.1.3 Sink Node

The sink node is the very last node in the CDFG. It has no outflow arcs. It is a virtual node only present to allow easy traversal of the CDFG. It does not result in an operation being generated in the final code sequence.

7.1.2 Arc Types

7.1.2.1 Data Arcs

A data arc represents the flow of data from the result of one operation to the operand of another. The data widths of the result and the operand do not have to be identical. If they differ then appropriate widening or narrowing is performed by the hardware.

When generating the final code the scheduler must examine each of the data flows represented by a data arc and arrange suitable transport of the data item from the generating unit to the consuming unit. The existence of a data arc between two operations guarantees that a physical data path exists between them if the CDFG is concrete. Data arcs within idealised CDFGs do not necessarily correspond to real connectivity.

Each data arc is annotated with a latency value. This represents the number of clock cycles between issuing the generating operation and the result becoming available. The scheduler ensures that sufficient distance is placed between the two operations that the result will be available. Moreover, the scheduler must ensure that the result is read before being overwritten by a subsequent operation issued to the same generating unit.

7.1.2.2 Control Arcs

A control arc represents an ordering constraint between two nodes in the CDFG. The dependee node cannot be issued before the dependent node. Control arcs are used to represent various scheduling constraints that are not associated with data flow. For instance, control arcs are generated between certain load and store memory operations whose ordering cannot be changed without affecting the program results.

Each control arc is annotated with a minimum distance value. This is the minimum number of clock cycles that must separate the two operations. A distance of 0 indicates that they can be issued on the same clock cycle.

7.1.2.3 Weak Arcs

Weak arcs also represent an ordering constraint between two nodes in the CDFG. If possible the scheduler avoids breaking the dependency. However, unlike a control arc, the dependency can be broken if necessary. For instance, the scheduler may have to break the dependency in order to obtain improved code density. If the dependency is broken then there is a consequence in that a conditional arc associated with a weak arc becomes active. If the weak arc dependency is honoured then the conditional arc is ignored. This mechanism allows compensation operations to be automatically inserted if a weak arc dependency is violated.

Each weak arc is annotated with a minimum distance value. This is the minimum number of clock cycles that must separate the two operations. A distance of 0 indicates that they can be issued on the same clock cycle.

7.1.2.4 Conditional Arcs

All conditional arcs are associated with particular weak arcs that emanate as an outflow from the same operation. Under normal circumstances a conditional arc is ignored and does not impose a constraint on operation ordering. If the dependency of the weak arc is violated then the conditional arc becomes active. This means that the dependency it represents must be honoured and the dependee operation must be issued. The dependee operation is only issued if one or more of the conditional arcs flowing into it become active. This mechanism allows compensation operations to be automatically inserted if a weak arc dependency is violated.

Each conditional arc is annotated with a minimum distance value. This is the minimum number of clock cycles that must separate the two operations. A distance of 0 indicates that they can be issued on the same clock cycle.

7.1.2.5 Tunnel Arcs

A tunnel arc forces a particular ordering between operations. A tunnel arc is used as a hint to the CDFG optimiser that the control arc is present because of a data item "tunneling" through the register file or memory. The data flow is not explicit but, instead, is stored in the internal state of the register file or memory unit. For instance, if a data item is written to a particular register and subsequently read by a later operation then a tunnel arc may be generated between the two operations. This indicates that a data item is being transferred between them and thus the read cannot happen until after the write is completed.

The CDFG optimiser may rewrite the CDFG surrounding a tunnel arc to provide a direct and explicit transfer of a data item if there is no particular reason why the register file or memory unit needs to be used. This forms part of the process of eliminating unnecessary register file accesses if data cannot be transferred directly between functional units.

Each tunnel arc is annotated with a minimum distance value. This is the minimum number of clock cycles that must separate the two operations. A distance of 0 indicates that they can be issued on the same clock cycle.

7.1.2.6 Association Arcs

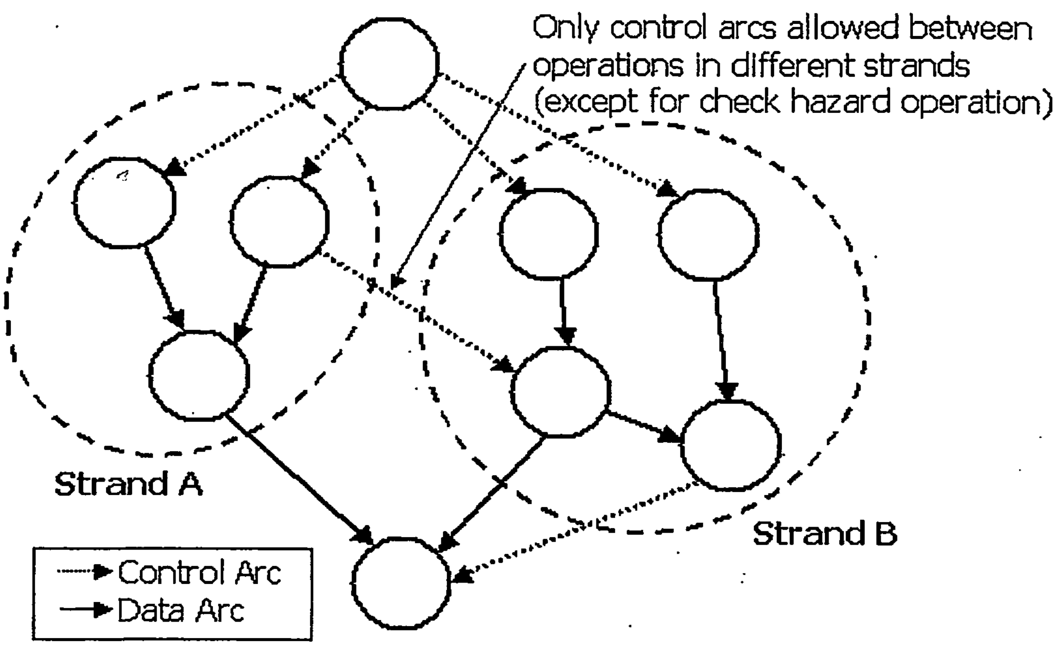
Association arcs are generated between operation nodes that read the same value from the register file or memory. Association arcs do not impose any ordering constraint between the nodes and during the scheduling process they are ignored. The CDFG

optimiser uses association arcs to identify opportunities where multiple reads could be eliminated in favour of a single read. In that case the single read can supply all the consumers of the required data.

7.1.3 Strand Sets

Each region is composed of a number of strands. All operations are a member of one particular strand. Strands are used to separate operations that belong to different control flow paths in the region. In general, strands correspond to basic blocks.

The following diagram illustrates a CDFG containing two different strands:



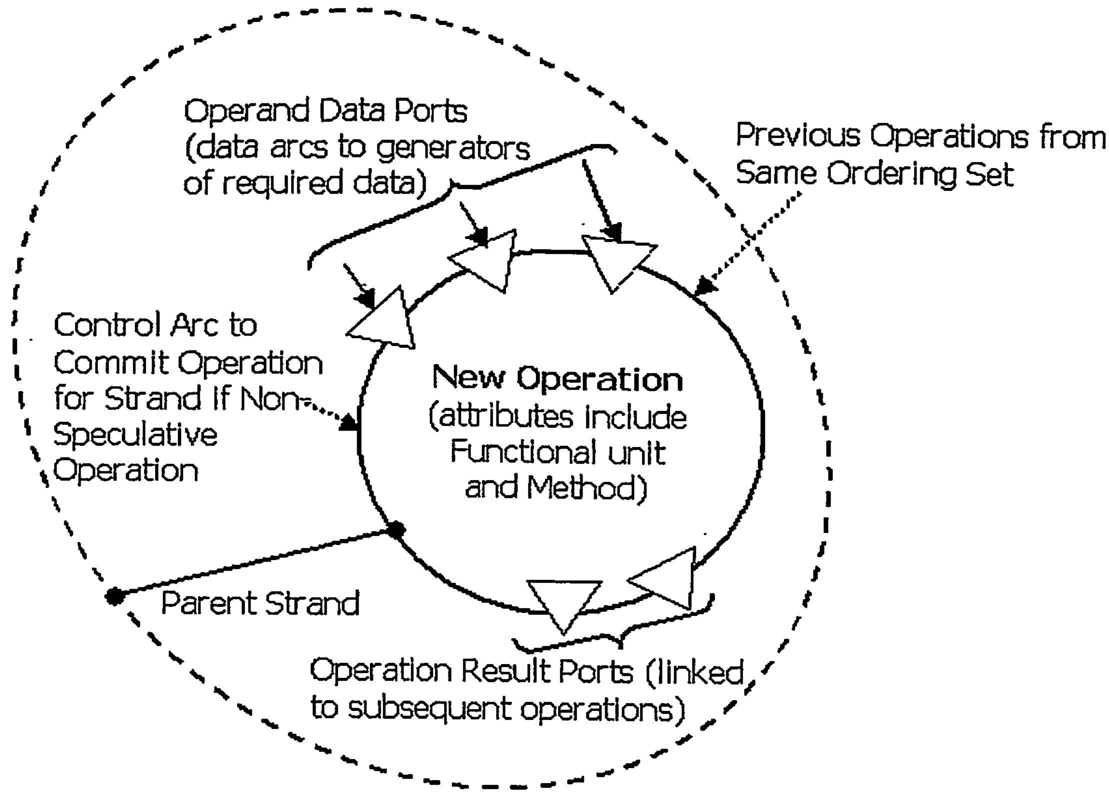
The data flow within a particular a particular strand must be self-contained. Data arcs between different strands are not legal (with the exception of the store address for a check hazard operation). This is because on any given execution of the region certain strands may be disabled. Thus a later strand may read an undefined value since the earlier strand will not have calculated the data item. Communication between the strands occurs through the register file and memory. All registers that are live at the end of a strand in the host code are written to the register file in the translated code. Thus subsequent strands can read the data values via the register file.

Control, weak, conditional and tunnel arcs are permissible between strands. In some circumstances the CDFG may rewrite a tunnel (between a register write in one strand and a register read in another) into a combined write/read register operation.

7.2 Creating a New CDFG Node

As instructions are translated, new operations are added to the CDFG. A single node in the CDFG represents each operation. A method for adding a new operation is identical for all types of operations. The new node has to be connected appropriately to other nodes in the CDFG to show the data flow and constraints on ordering of operations.

The diagram below illustrates the arc connectivity associated with a new node:



The new node has a number of associated attributes that are dependent upon the type of operation that the node represents. All operation nodes have an associated functional unit and method indicating how the operation is to be performed on the hardware. Squash operations also have an attribute of the strands that they control.

Each new operation is associated with a particular strand. During translation there is a current strand to which all new operations are added. The link to the parent strand is used when generated the final code to determine the strand number to be associated with the operation.

An operation has a number of input data operands. Each of these may have user definable data widths. A data flow arc is connected to the preceding operation that generates data for the operand. In this manner the data flow in the program is elucidated. Each operand has only a single data flow arc as all data is generated from within the same strand in which the control flow is linear. The only exception is the operand of a check hazard operation, although the data still emanates from a single source in a previous strand. If there is any confluence of potential data values form the control flow topology then this is hidden by using the register file. Thus the data can only come from a single register read operation in the same strand (although the data may have tunneled from more than one potential register write). Each data arc is annotated with the latency of the functional unit that is to calculate the value. This is used by subsequent critical path analysis of the CDFG to help determine the best order to issue operations in.

If the operation cannot be performed speculatively then a control arc is generated form the commit operation for the current strand. This ensures that the operation cannot be issued before the commit and thus must be issued in the committed phase of the strand. Whether a method can be issued speculatively or not is an attribute detailed in the hardware configuration file.

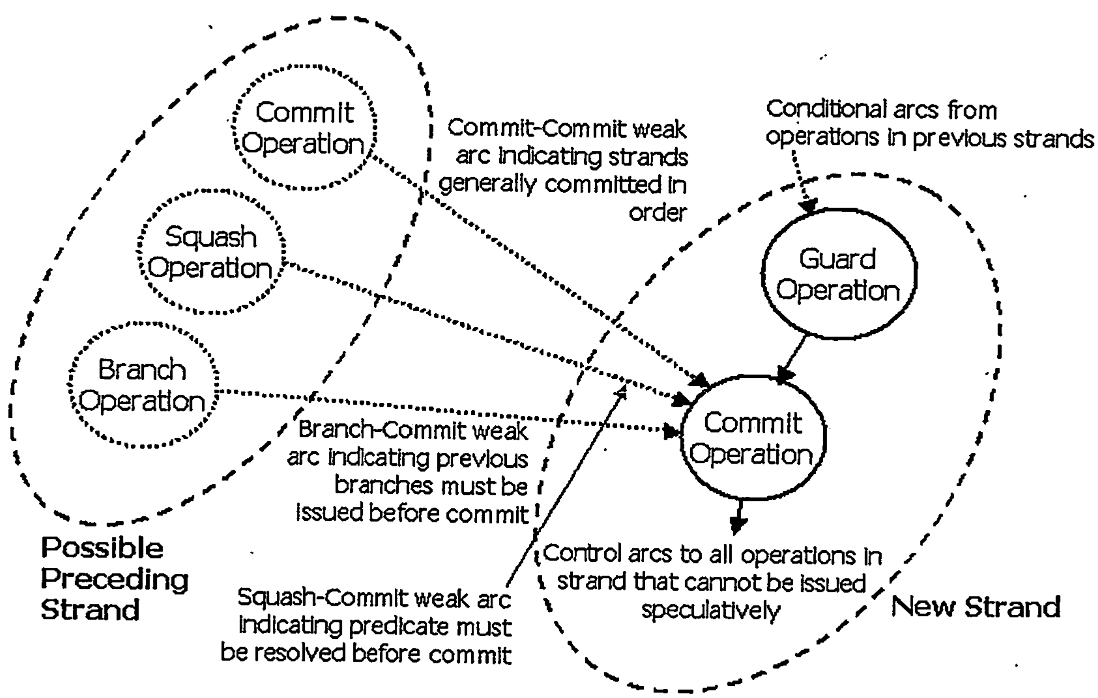
Control arcs are also issued to preceding operations that are in the same dependence set as the new operation. Dependence sets are used to provide an enforced ordering between certain methods that must be issued in a particular order. If the operation is in the same strand then a control arc is used. If the operation is in a preceding strand that is reachable from the current strand then a weak arc is generated (unless the function being translated is critical). An associated conditional arc is generated to the guard node for the strand so that if the ordering is violated then the guard for the strand is enabled. The arcs are annotated with the blockage value for the method used in the preceding node. This ensures that a suitable number of clock cycles are left between functional unit usages.

The operation output ports are subsequently connected to one or more operations that use the result. Results from an operation do not have to be used, in which case no data arc is connected to the result port. However, each operation must have at least one successor arc. This can be to the CDFG sink node if required.

7.3 Creating a New Strand

A region is composed of one or more strands. This section describes the additional operations nodes that need to be added to the CDFG whenever a new strand is started. New strands may be started for a number of reasons but their initiation is normally associated with the start of a new basic block in the translation. The strand mechanism allows multiple basic blocks to be represented in a single region and to be optimised and scheduled as a single entity.

The diagram below shows the additional operations created at the start of a new strand:



Two operations are created in the new strand. Firstly, there is a guard operation to act as a sentinel for entry to the committed phase of the strand. The guard operation is a conditional node and is only actually issued if a weak arc dependency between an operation in the strand, and some preceding strand, is violated. Secondly, a commit operation is issued. The commit operation represents the phase transition barrier between the speculative and committed phases of the strand.

A number of weak arcs are generated between operations in any preceding strands and the new commit operation. If a critical function is being translated then some of these arcs may be standard control arcs to fully enforce the ordering. An arc is generated to the commit operation of a preceding strand. This ensures that strands are generally committed in ascending order. An arc is generated to the squash operations that control the predicate for the strand. There may be a number of these squashes if the new strand is in a nested control flow area. The arc ensures that all potential squashes associated with the strand are evaluated before the committed phase of the strand is entered. An arc is generated to the branches in previous strands. This ensures that if a branch is to be performed by a previous strand (and thus squash the new strand) then the branch has been issued before the committed phase of the new strand is entered.

All operations that cannot be issued speculatively within the new strand have a control flow arc connecting them to the commit operation. This ensures that they are not issued before the commit phase is entered. Other operations do not have this dependency and migrate to earlier then the commit operation in the schedule and become speculative.

Each new strand that is created has an associated static execution condition. This corresponds to the state of the condition codes in the host architecture. This is used to support the translation of conditionally executed instructions. A new strand needs to be started whenever there is a conditional instruction and the condition does not correspond to the condition for the current strand. If there is any write to the condition code then the condition of the current strand is reset.

7.4 Liveness Analysis

As part of the region construction a full liveness analysis is performed. This provides the set of registers live on entry to and exit from each instruction. This information is stored in a data structure allowing lookup of live register information on the basis of the host instruction address. An iterative data flow convergence technique is used to calculate the liveness.

The liveness analysis covers the condition code registers as well, treating each condition code as a separate register.

All function calls (both direct and indirect) are considered to modify all volatile registers. This is consistent with the Application Binary Interface (ABI) of the host processor. Hand written assembly functions that violate these rules cannot be reliably translated. A function must not corrupt non-volatile registers. Moreover, code cannot rely on volatile registers being preserved across function calls. All code to be translated is expected to conform to the ABI.

The live-in register set for a function call is dependent upon the parameters used by the called function. Analysis is performed on each function to determine which parameter registers it reads. Whenever a call is made to that function that information is used to form the live-in set for the call itself. Indirect function calls are assumed to have all parameter registers as live. If the parameter set used by a function changes then that will cause all code that calls the function to be re-translated.

8 Functional Unit Allocation

8.1 Philosophy

The unit allocation operates on the idealised CDFG generated in the previous step. The purpose of this stage is to allocate concrete units where operations are being performed that could be executed by a number of different units.

Each node in the CDFG is visited. If a single unit can only perform a node operation then the selection process simply selects that unit. A concrete unit is then allocated to all other nodes. The ordering is based on the number of nodes of the same type that are predecessors or successors of the nodes in the graph. The nodes with the greatest number of such predecessors/successors are processed first. This ensures that the nodes that will have the most influence on the allocation of other nodes are handled earlier. If a particular use of the unit is intrinsically ordered by the existing data flows in the graph then no subsequent latency adjustments need to be made when trying to allocate the same unit.

If a unit has multiple ports (such as a register file or memory unit) then the unit allocation also selects which particular port is to be used. In effect each port is treated like an independent unit.

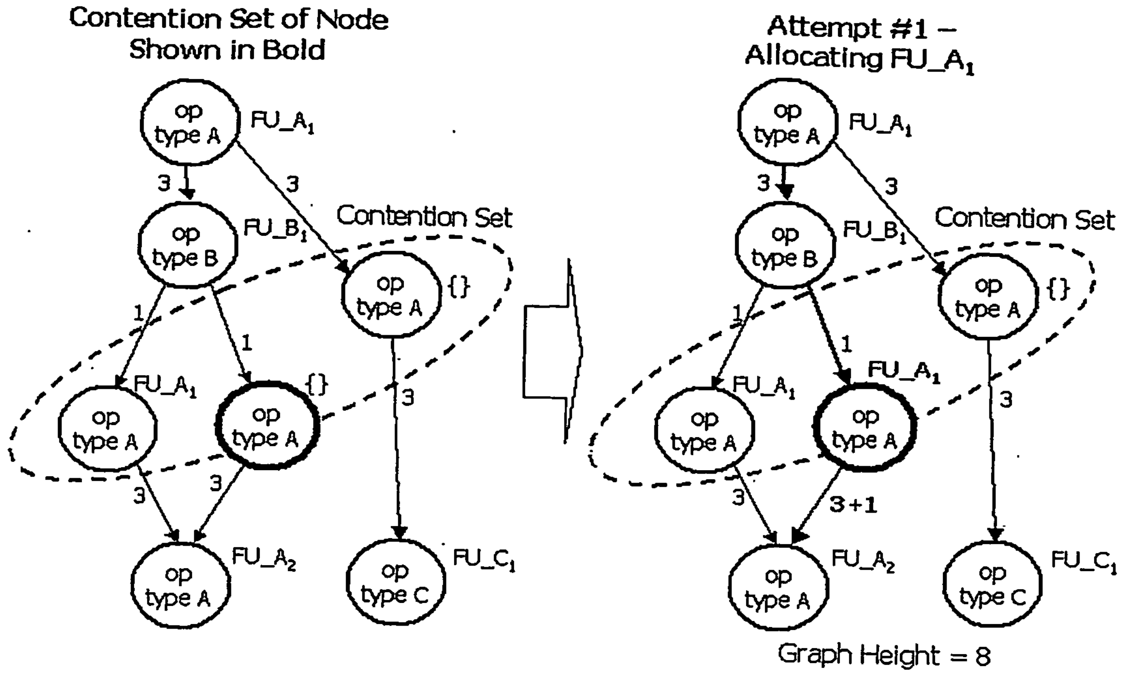
The unit allocations are remembered in terms of the sequence order that the operations were added to the graph. These are then used during the transported CDFG construction to allocate the correct units.

8.2 Conflict Adjustment

The purpose of the conflict adjustment is to measure whether there is a potential for the same unit to be required in parallel with the candidate allocation being tested. Uses of the unit that are definitely before or after the current allocation (as determined by the graph dependencies) do not impact the usage of the unit. However, if the unit may be used in a section of the CDFG that could be scheduled in parallel with the candidate usage then that could impact parallelism. If the potentially parallel nodes were to be allocated to different units then they could potentially be issued on the same cycle. If they are allocated the same unit then that is not possible. The purpose of the conflict adjustment is to modify the latencies within the CDFG to reflect this possible degradation in parallelism. The allocation that maximises the chances of parallelism is then selected.

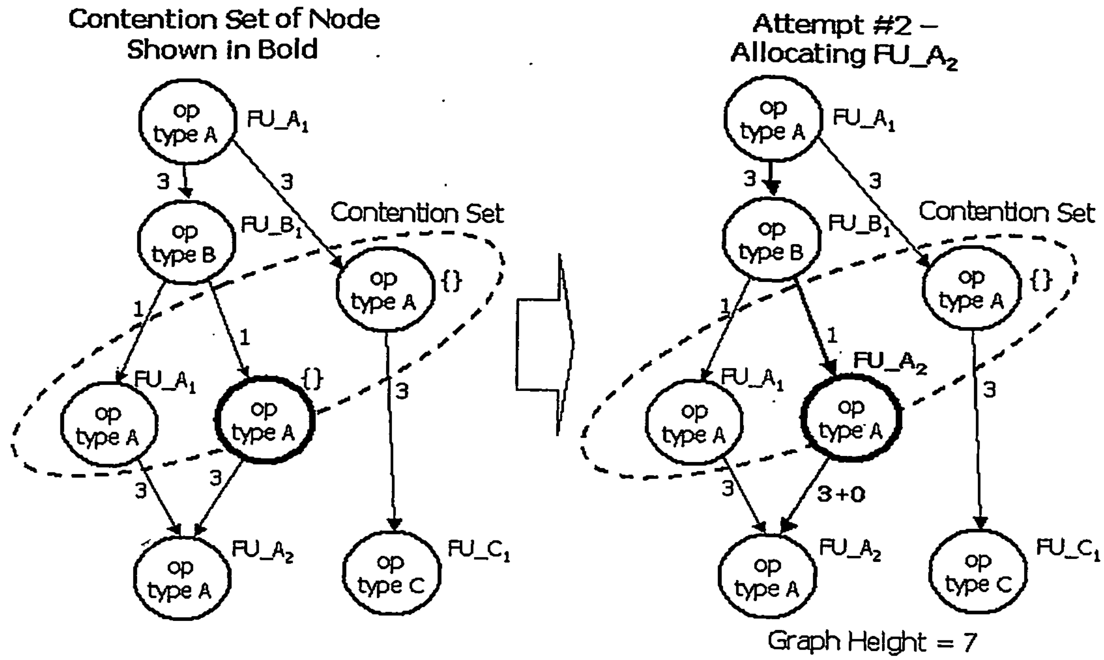
Selection is based on an augmented graph height analysis. Each possible unit is selected in turn and the resultant graph height calculated. The unit giving the lowest graph height is selected. If there are multiple units resulting in the same graph height then the lowest numbered unit is chosen. The output latencies for the selected node are adjusted to account for potential serialisation caused by use of the same unit. This adjustment factor is based on the number of uses of the same unit in the graph that are not forced predecessors or successors on the basis of the graph dependencies (determined from a transitive closure matrix). The adjustment is the blockage of the unit multiplied by the number of such potentially parallel uses of the unit. This mechanism thus adds an extra delay where parallelism may be restricted by the use of the same unit. This methodology tends to allocate different units for calculations that can be performed in parallel according to the idealised dataflow graph.

The diagram below shows the modifications performed to the CDFG on the basis of unit contention:



The original CDFG is shown on the left hand side. The node that is outlined in bold is the current one for which unit allocation is being performed. The dashed line represents the contention set for the node. These are all the nodes of the same type (and could thus be allocated to the same unit) that are parallel to the candidate node in the CDFG. That is, depending upon the schedule produced, those operations could be issued in parallel to the candidate node. There are two other operations of type A in the contention set. One is currently unallocated to a unit (and thus ignored) while the other has already been allocated to FU_A_1. In the first allocation attempt the first unit of type A (FU_A_1) is allocated. Since the unit is used in the contention set an additional latency of 1 is added to the output arcs of the candidate node. This represents the blockage of the unit and this the potential delay caused by serialisation with the existing allocation. When the graph height analysis is performed the total height is 8.

The next step is to try a different allocation of FU_A_2 to the candidate node:



That unit is not used within the contention set so no latency adjustment is required. This leads to a graph height of 7. Since this is lower than the previous allocation then it is selected in preference.

Note, however, if that unit allocation is being performed after unit placement has been completed in the architectural creation then adjustments for transport distance as detailed below are also added. They could also influence the allocation decision.

8.3 Transport Adjustment

If the unit placement has already been performed then a further layer of adjustment is performed on the arcs associated with the node being allocated. The principle is to add additional latencies to inflow and outflow arcs that reflect the likely transport costs. The transport cost adjustment is based purely on the distance between units rather than their actual connectivity in order to produce stable unit allocations around which specialised connections can be optimised. Thus once the unit placement has been performed the unit allocations will remain constant for the same code. The Euclidian distance to the required unit is calculated. For each arc there is a distance below which the cost adjustment is considered to be 0. This is the maximum direct connectivity distance (a general optimisation constant). If the Euclidian distance is below that then no adjustment is added, otherwise the adjustment is the Euclidian distance minus the zero cost distance. If the producer/consumer unit is fixed then the exact position of the target is known. If the producer/consumer is unallocated then the closest unit of the appropriate type is used. The transport cost adjustment has the maximum impact on the most critical arcs to and from a node. This mechanism attempts to allocate units within clusters that have appropriate local functional units.

The diagram below shows an example transport adjustment:

The node has been allocated to functional unit FU_A_2. The outflow arc is to the same node again so is within the zero cost zone. The input, however, is from node FU_B_1 which is outside the zero cost zone. The corresponding arc is thus augmented which an additional latency of 1 clock cycle. This represents the distance of the unit outside of the zero cost zone. The additional latency increases the graph height to 8 clock cycles. Thus the previous allocation is selected in preference to this one.

9 Transport Allocation

9.1 Philosophy

Transport allocation occurs as part of the construction of the CDFG. The low level functions for implementing the addition of new operations into the CDFG also perform transport allocations. In this way the transport allocation can be logically seperated from the task of instruction translation from the source binary.

The unit allocations determined during the unit allocation phase are utilised. The resultant CDFG can be scheduled directly for use as the shadow code. This phase is not performed prior to unit placement. A later transport optimisation phase allows unnecessary main register read and write operations to be removed in a way that ensures that the resultant CDFG can always be scheduled without deadlock.

All execution units may have multiple output registers, each forming a point-to-point connection with an operand of another execution unit. Thus as new connections are added new registers are also added automatically. A field is included in all operations to select which output registers are updated. A bit mask rather than selector is used to allow efficient support for multiple consumers of a data item that are reached through different routes and thus different connections.

Transport allocation is performed as part of the overall translation process. The transports are built as part of the CDFG construction so cyclic graphs cannot be generated. As particular data values are calculated they are then transported to the appropriate operand of the unit on which they will be used. If there is a direct connection between the result port of the unit and the required operand then no additional operations are required. In other cases addition copy operations are generated to transport the data item to the required operand. A search is performed from the output to all connected nodes to find the best route to the destination operand. The route with the shortest latency is always chosen.

On each occasion an operation is added to the CDFG (including copy operations) addition arcs may be added to force an order on the use of the associated output register. This forces a serialisation on the use of the output register resources and prevents live data values from being overwritten. The register resource structure is used for this purpose (it is also used to provide ordering on main register accesses). When an operation is generated dependency arcs are added to all previous readers of the output register. The new write forms a new live range for the register that is held in the register resource structure. This ensures that the operation is not scheduled until all previous reads of the previous value are complete. The arc latency may be negative as the writing operation may commence before the read has completed – it must just happen before the output register is overwritten at the end of the pipeline.

When a transport is performed the last register written in the chain is kept live. This prevents the register being used in any subsequent transports until the value is consumed. The value is made dead at the point of consumption. The reachability analysis ensures that all operands for a particular functional unit are reachable without having to

use the unit itself to perform copy operations. Thus as long all live values are operands of the same functional unit then code can definitely be generated. Consequently the locking of operands to multiple functional units (or holding an operand live for a functional unit over a preceding use of it) is not legal in the code generation. In some cases this is a requirement. The affecting operands are made exempt from the paths used by the reachability analysis. No attempt is made to allocate arcs to check hazard operations (they are disabled) as they are inserted earlier in the CDFG.

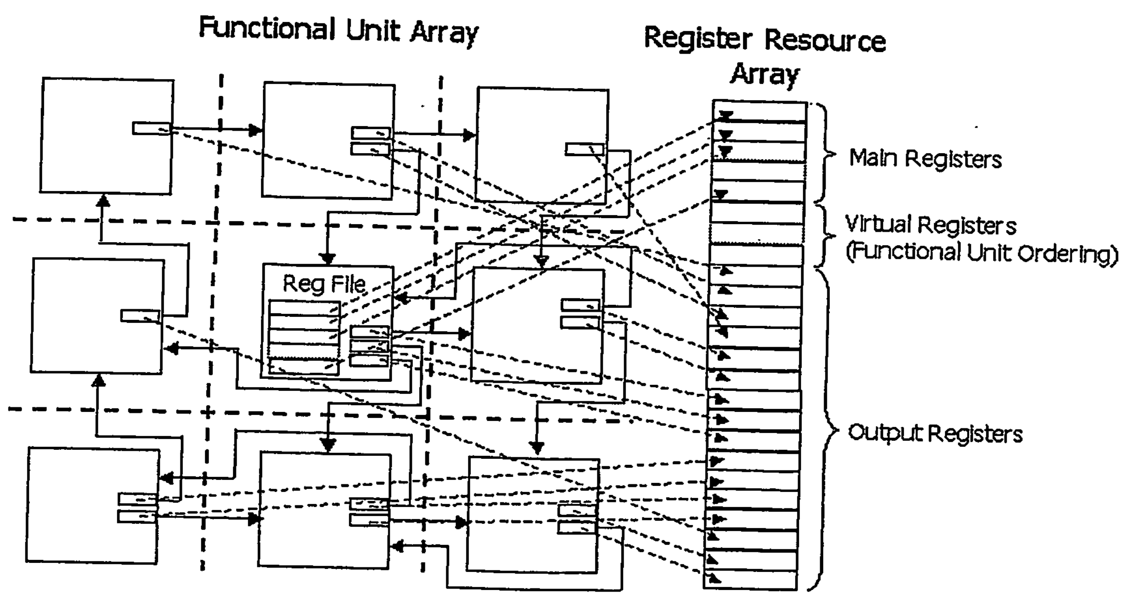
9.2 DU-Chain Construction

This section describes the dependence flow construction in the CDFG. The depedence flow is effectively the Definition-Use chain constructed for each of the registers in the system. The dependence flow constrains the order in which operations may be scheduled. Dependence flow is maintained using arcs between operations in the CDFG. The dependence flow is distinct from the dataflow of operands and results between operations and provides an additional level of constraint in terms of operation scheduling.

The main dependence flow is associated with register file usage. Certain operation orderings are required in order to ensure that correct values are read and written to the register file. The user may also define explicit ordering constraints between operations. These are defined in the hardware configuration file in terms of read and write dependence sets for each method. Common mechanisms are used for storing state information for both these and register file dependencies. Dependencies for memory access operations are dealt with separately.

9.3 Architectural Register Resources

The diagram below shows the register resources that are present in the architecture. Each of these is tracked as part of the DU-chain construction.



The diagram illustrates the register resource array in which the DU-chains are constructed on the right hand side. Each entry in the uniform array corresponds to a register resource in the architecture. The dotted lines show example mappings from those resources to the corresponding entries in the register array.

The different types of register resource are as follows:

- **Main Registers:** The set of main registers that are equivalent to those in the source architecture.
- **Output Registers:** The output registers present in the connectivity network of the functional units. Each such register is mapped to an entry ion the register resource array.
- **Virtual Registers:** There are a number of virtual registers that are used to provide orderings between particular methods. The method definitions allow it to effectively read or write to a set of virtual registers. This allows dependencies to be created between different methods so that their order is not changed. For instance a new functional unit may be defined which has internal state and whose methods must be issued in a certain order. Use of virtual registers allows those dependencies to be make explicit in the CDFG.

9.4 DU-Chain Structure

During the construction of the CDFG a register definition structure is maintained. This is an array of objects, each holding status information for one of the registers in the register file. It lists the set of write operations for a particular register that reach the current position. It also lists the operations that have subsequently read the contents of the register. All register file accesses are performed using known register numbers (no indirect access is possible) so the dependencies between operations is fully known.

The register definition structure is maintained as reads and writes are issued to the register file as part of the translated set of operations. It allows suitable dependency arcs to be generated between register file accesses that maintain the correct code semantics. The dependencies do not restrict the relative migration of accesses to different registers.

9.5 Virtual Registers

Ordering constraints associated with explicit dependence sets are handled using virtual registers in the register definition structure. This allows common mechanisms to be used for handling both types of dependencies. Each dependence set is allocated a particular virtual register.

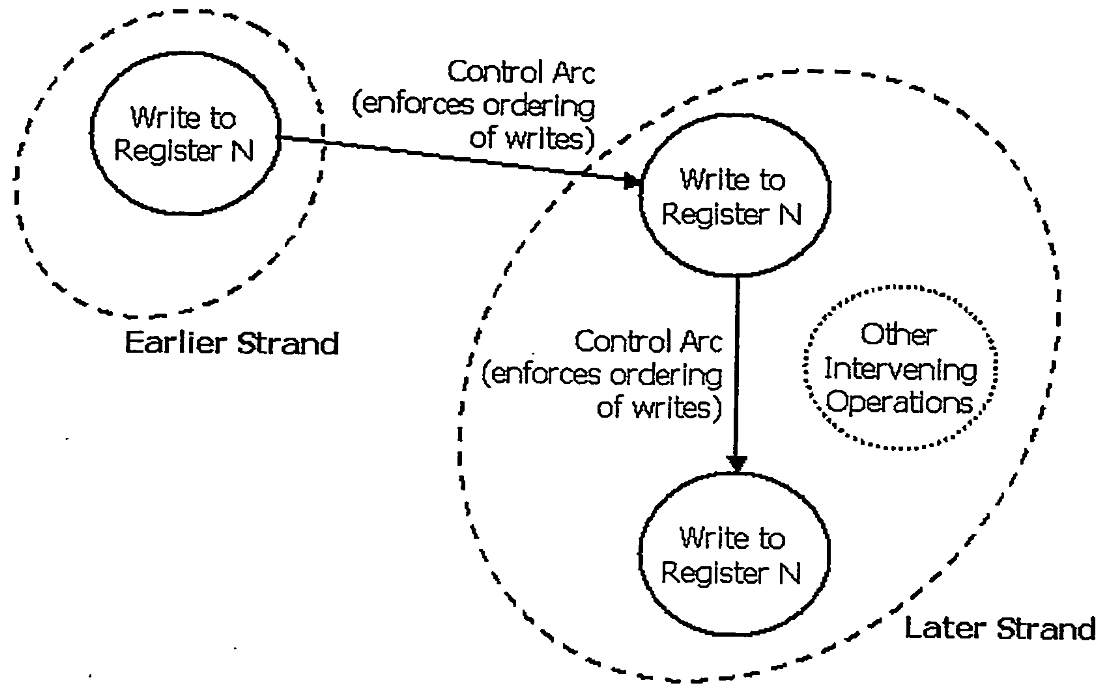
A method may be a member of a number of writing dependence sets. The system acts as though the operation performs a write to each of the virtual registers associated with those sets. The appropriate dependence arcs are generated.

A method may also be a member of a number of reading dependence sets. The system acts as though the operation performs a read form each of the virtual registers associated with those sets. The appropriate dependence arcs are generated. Note, however, that tunnel arcs are not generated.

9.6 Register Writes

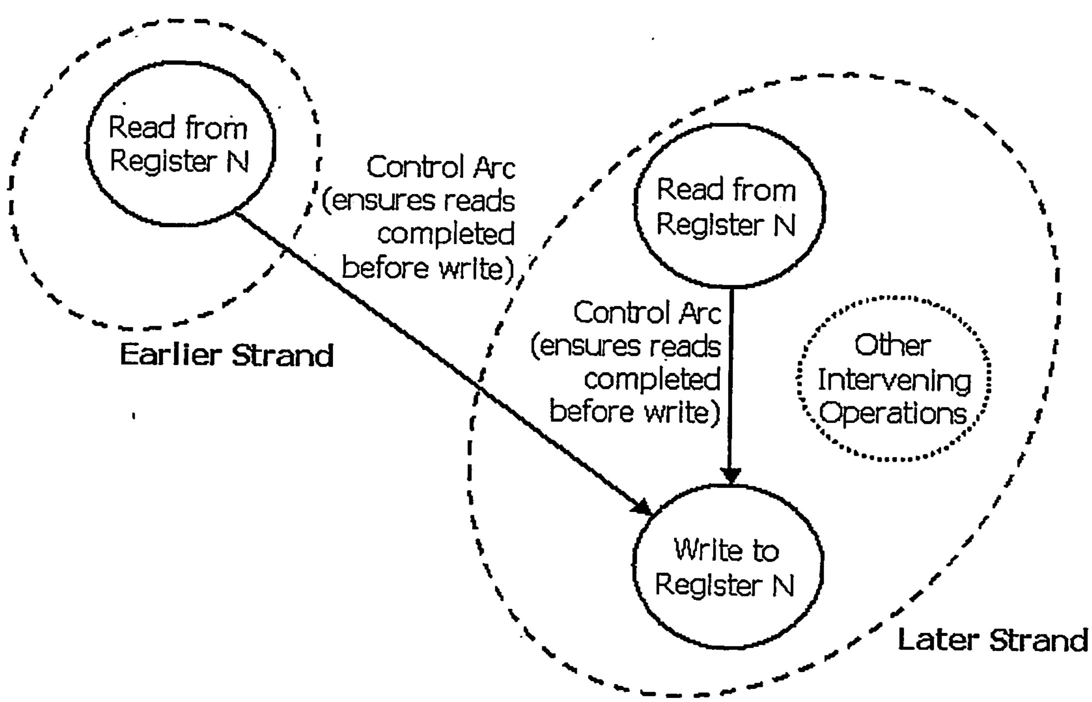
Whenever a register write is performed the last writing operation is updated in the register definition structure. However, before issuing the register write operation, arcs are generated to preceding accesses to the same register. This ensures that operations are performed in an order that maintains the codes semantics.

Firstly, arcs are generated to the preceding writes to the same register as detailed in the diagram below:



These control arcs serialize the write operations to the same register. Within the same strand, control arcs are always used. Dependencies between strands use control arcs if a critical function is being translated. However, in other circumstances a weak arc may be used with a conditional arc to the guard of the later strand. This causes the later strand to be executed during a subsequent region re-execution if the dependencies are violated.

Secondly, arcs are generated to the preceding reads of the same register as illustrated in the diagram below:



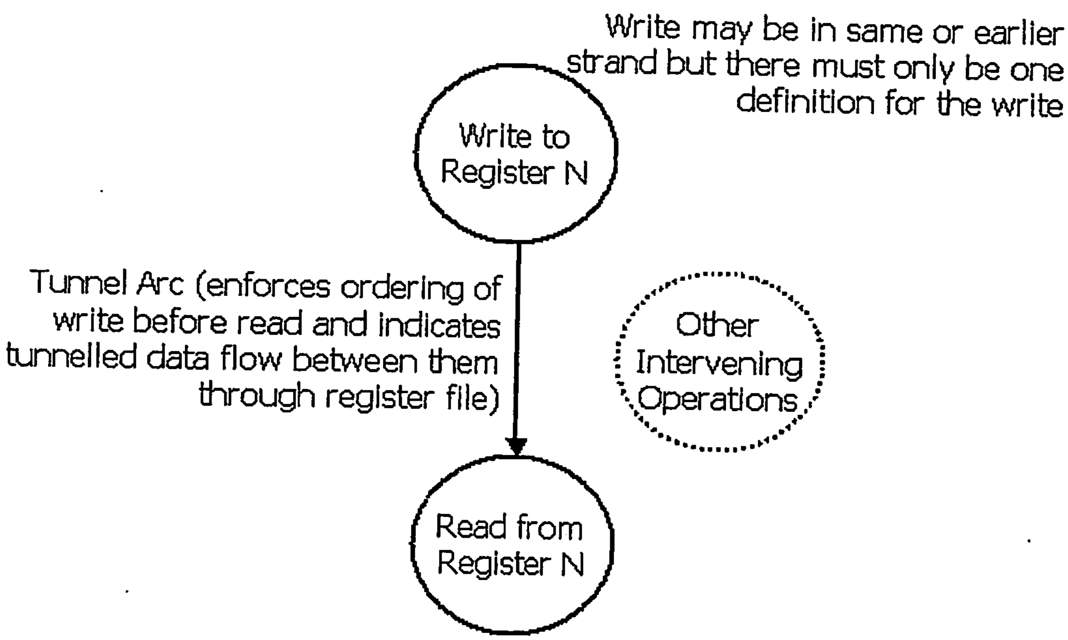
These arcs ensure that a write to a register is not performed until all reads of the previous value in the register have been completed. Individual arcs are created from each read to the subsequent write. This avoids serialization of the reads themselves, which can be freely reordering. Within the same strand control arcs are always used. Dependencies between strands use control arcs if a critical function is being translated. However, in other circumstances a weak arc may be used with a conditional arc to the guard of the later strand. This causes the later strand to be executed during a subsequent region re-execution if the dependencies are violated.

Each write to a particular register clears the list of read operations that access the register's previous value.

9.7 Register Reads

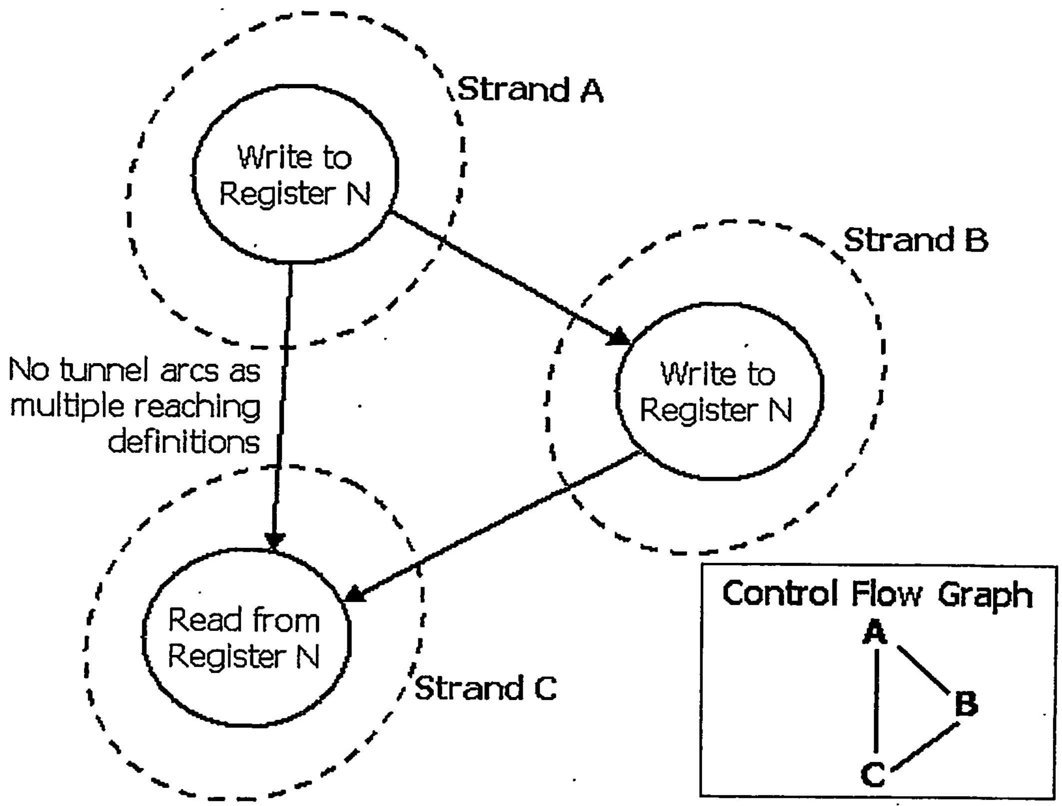
Whenever a read is performed on a particular register, the read operation is added to a list of reads associated with the register. This list is maintained in the register definition structure. This list is used to add appropriate arcs to any subsequent write to the same register to ensure that the write occurs after the reads have been completed.

When a read operation is generated an arc is generated to any preceding write to the same register, as illustrated on the diagram below:



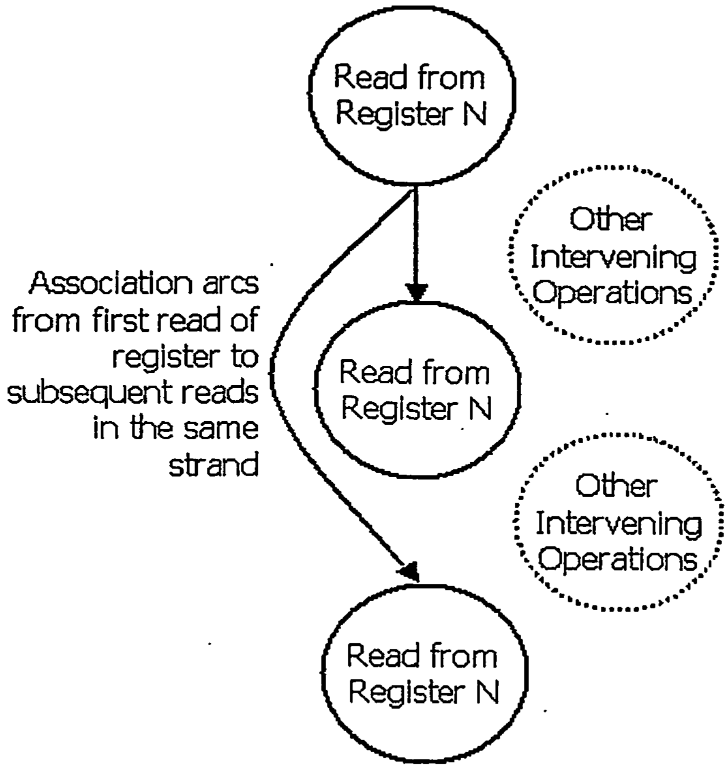
If there is no preceding write (i.e. the read is using a value stored in a previous region) then no arc is generated. If there is a single reaching write then a tunnel arc is generated to it. A tunnel arc indicates that data is being "tunneled" through the register file from the write to the read. The arc acts as a control flow arc in terms of maintaining dependencies but is a hint to the CDFG optimiser that the write and read could be eliminated in some circumstances and the tunneled data made explicit as a data flow arc. The register write may be in a previous strand.

In some circumstances there may be multiple reaching writes for a register. This can happen when there is a confluence of multiply control flow paths, as illustrated in the diagram below:



The diagram shows an IF-THEN construct where a particular register is written before the IF and in the THEN branch. If a read is subsequently performed then the data could be generated by either of the register writes. In this case control flow arcs are generated to the list of reaching write operations. A tunnel arc is not generated, as this construct is not amenable to subsequent CDFG optimisation.

Association arcs are created between the reads of the same register value within the same strand. Association arcs to not impose any ordering on the operations in the CDFG but are used to associate read operations that obtain the same value from the register file. The CDFG optimiser uses association arcs to eliminate multiple reads of the same register value. The diagram below shows three reads of the same register value and the association arcs generated:

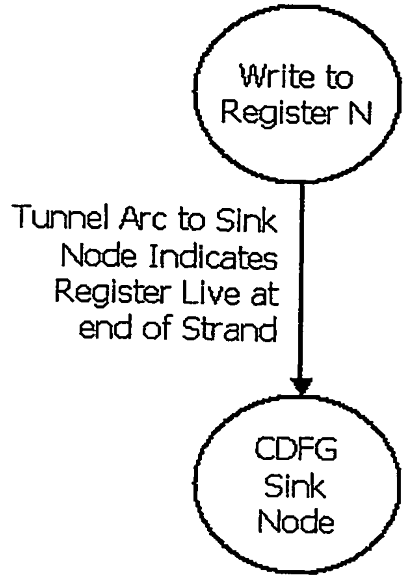


The register definition structure maintains a record of the first read operation for each register in the current strand. This information is used to create appropriate association arcs as further reads from the same register are generated as part of the translation process.

9.8 Externally Live Registers

If a particular register is live at the end of a strand then the written value must be maintained in the register. This is because it may be used during the execution of a subsequent region. The register liveness is determined from the full liveness analysis performed on the function being translated.

An externally live register has a tunnel arc generated to the sink node of the CDFG as illustrated below:



This tunnel node indicates that the sink (in effect the following regions) use the register value and it cannot be optimised away. The existence of the arc prevents the CDFG optimiser from removing the register write if it can rewrite the CDFG to use direct data flow.

Note that the same register can be "sunk" to the sink node several times in the same region. This is because different values of the register can be live at the end of different strands within the region.

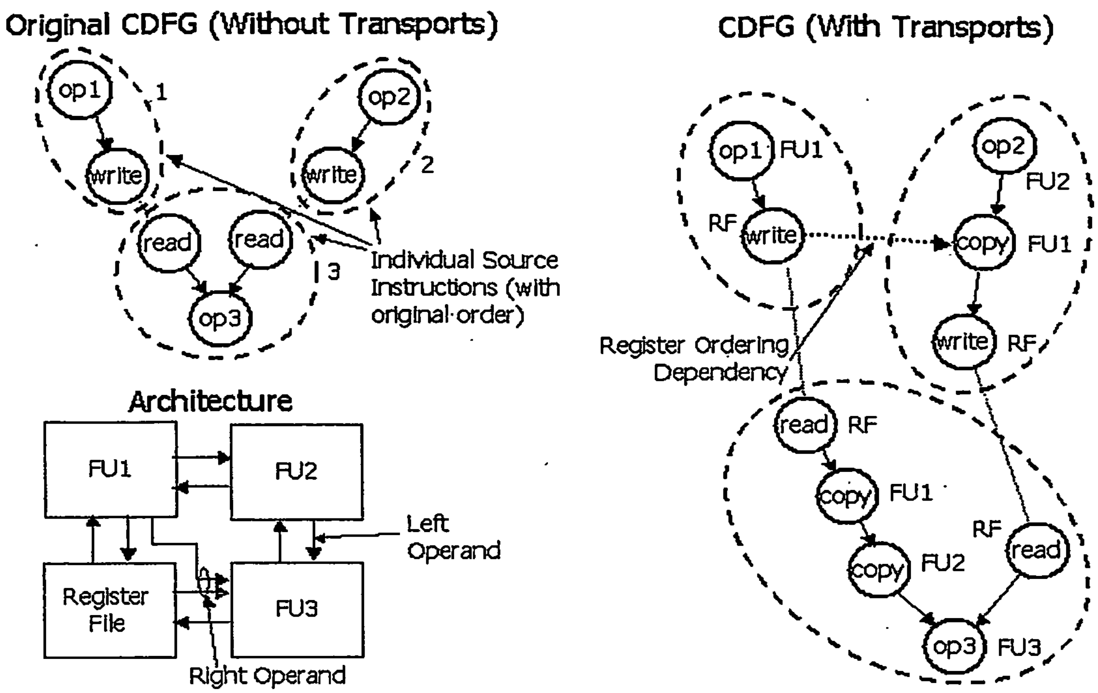
9.9 The Requirement for Transports

The constructed CDFG describes the operations in the original source program and the dependencies between them. However, the CDFG must also be extended to incorporate data transports between operations. Unlike traditional processors, the CriticalBlue processor is not a fully connected machine. Thus data items cannot be arbitrarily copied from one functional unit to another. If a bus does not directly connect two functional units then additional operations must be generated to move the data item. The connection has to be between the output result port of one functional unit and the input operand port of another. These additional operations must be scheduled like any other operation that is to be run on the processor.

The additional nodes are called copy operations. They simply copy the input of a functional unit to its output without performing any operation. Any functional unit is able to operate in copy mode whereby a particular input operand is selected and copied to all the result ports. The latency of such a copy operation is identical to that for ordinary operations performed by the unit, in order to simplify the scheduling problem when handling a mixture of both real and copy operations on a unit. The transport allocation algorithms choose a particular route that is to be taken by a data item from the source to the destination, generating copy operations on the intervening functional units.

During the transport allocation the route chosen is fixed and always represents a route with minimum latency through the connectivity network. The shortest route between each result port and each operand is pre-selected as part of the architectural optimisation.

Transport allocation is performed as the CDFG is being constructed. For illustrative purposes, the diagram below shows an example CDFG without and then with transport operations added:



The architecture of the simple example processor is shown on the bottom left hand corner of the diagram. As can be seen the functional units are not fully connected and this requires the use of some transport allocation copies in a complete CDFG. The CDFG on the left hand side shows the nodes as generated from three different source instructions. The first two perform an operation and write the result back to the register file. The last instruction reads those registers and performs another operation. The operations are not bound to particular functional units at that stage.

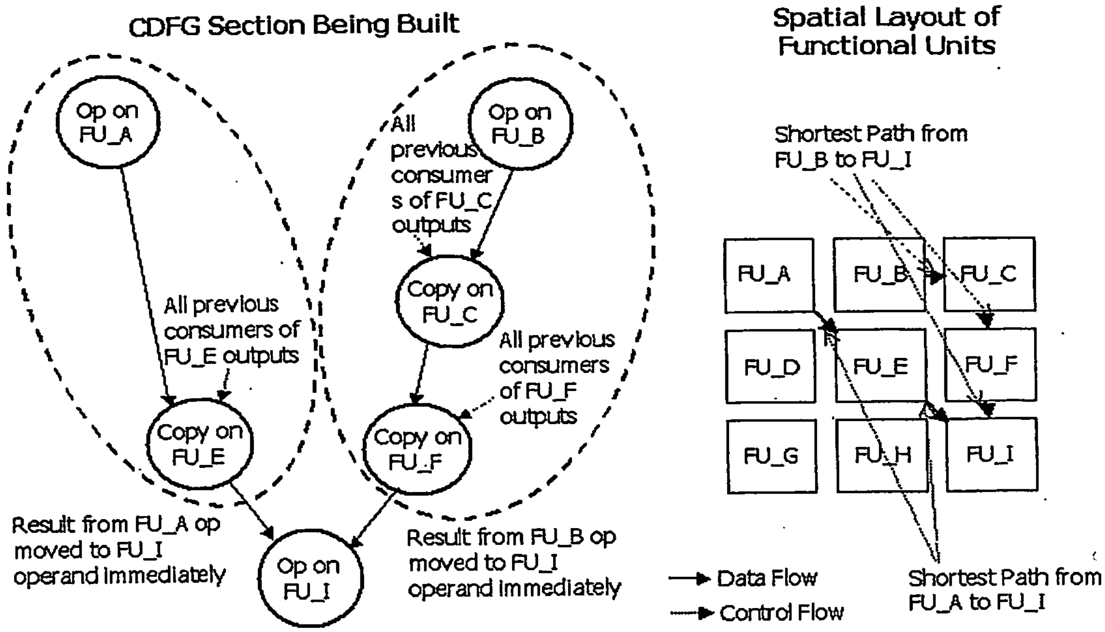
The CDFG on the right hand side shows the same CDFG with the required copy operations added. The op1 is bound to FU1 that can write directly to the register file. Thus no copy operation is required. The op2 is bound to FU2 that is not connected to the register file. A copy operation through FU1 is added. The copy is dependent on the completion of the earlier register file write (i.e. the consumer of the last use of the register in FU1). Thus the copy cannot be completed until the previous use of the output register has been completed. Register ordering dependencies prevent the register file reads being scheduled earlier than the register file writes. Two copy operations are required to move the required data to the first operand of FU3.

9.10 Building Transports

The required operations to perform transports of data are added to the CDFG as part of its construction process. As individual instructions are translated from the input binary they are converted into sequence of individual operations, as described previously. Most instructions consist of some operations to load the data from the required registers in the register file, the actual operation itself and then followed by an operation to write the result back into the register file.

In general it cannot be assumed that it is possible for the results from a register file read to be directly accessible by the functional unit that is to perform the instruction operation.

That is, the units may not be directly connected. Thus whenever data must be transferred between arbitrary functional units some transport allocation may be required. The diagram below shows an example of transport allocation:



The diagram on the right hand side shows the spatial layout of the processor. An operation in unit FU_I needs to be performed on results generated from units FU_A and FU_B. The arrows in the diagram show how data is transported across the array to the required destination.

The corresponding CDFG is shown on the left hand side. When the operation on FU_A is added to the CDFG a copy operation is also added to move the result to the appropriate operand input of the FU_I operation. All operations are added to the CDFG using lookahead transport allocation so that the operations are issued to move the result to the required consumer. Since all buses in the architecture are single source and single sink this means that the data item is moved to the appropriate output register to drive it onto the required operand. Thus the requirement to move subsequent operands for the same operation is guaranteed not to impact the data item. As copy operations are issued and required dependencies are created to ensure that the copy is issued after any previous uses of the same output registers. Since all dependencies are created to existing entries in the CDFG the construction is guaranteed to lead to an acyclic graph.

The movement of the data item from the result of FU_B to the other operand of FU_I requires two copy operations.

10 Transport Optimisation

10.1 Philosophy

The transport optimisation pass visits each of the arcs in the CDFG to allocate them a new route if that can improve parallelism. The allocation to a particular execution unit where there are multiple execution units of the same type may also be modified. This pass also performs register optimisations and builds a connectivity request matrix for adding new connections to the architecture. All shadow operations are also removed at this stage.

The optimisation is done in order of arc criticality with the most critical paths being optimised first. Critical path analysis is redone after each change to the allocation but any single arc can only be optimised once. This ensures that the most critical paths are given the first choice of transports. The complete path through copy operations is considered to be a single path for optimisation purposes (as the optimisation aims to change the copy operations).

An optimised CDFG is only valid if the new arcs can be added with causing the graph to become cyclic. A matrix is generated showing the transitive closure of the CDFG. When a new arc is added a test is made to see if it makes the graph cyclic.

Register optimisation is included within the transport allocation phase. Optimisations are only performed if transport can be provided without deadlock. If an arc is from a register read then an initial transport attempt is made to feed from the source for the register write (if there is a tunnel connection to the read) to the item. If allocation for that fails then the transport is scheduled from the first read of the same register value. If that fails then the actual CDFG arc is scheduled. Use counters delete unused register reads and writes if bypasses are successful. Transport allocation of data to check hazard operation may fail in which case the check hazard is deleted and the associated weak arc is made strong.

Register optimisations can freely occur within a strand. Register optimisations may also occur between a dominating strand and a consumer strand that it dominates. When such an optimisation occurs all weak arcs between the two strands (and any intervening strands) are made strong. This obviously has an impact on the resultant graph height and thus whether the optimisation is to be kept. Forcing linking arcs to be strong ensures that no region restarts occur between the strands and thus the dominated strand will be executed on the same iteration as the dominating strand.

Before a new allocation is attempted the existing one is removed (along with any copy operations). It is reinstated if no better path can be found. The finding of legal paths is a complex optimisation problem with an extremely large search space so a number of heuristics are employed. The basic scheme is to perform a depth first traversal of all routes forward from the result port. At each stage an attempt is made to use a direct route from the producer to the consumer. Visit flags are maintained so that no attempt is made to follow the same route more than once during the traversal.

At each stage (including the initial output from the producer) the output register write has to be inserted into the live range for the register. An insertion attempt is made at each point. As the write is inserted the appropriate arcs are added to ensure that the write occurs after previous reads and the reads and performed before the next write. If that leads to deadlock then the insertion point is discarded. Once all insertion points are attempted then the one with the one leading to the lowest graph height is selected. If there are multiple insertion points leading to the same graph height then the one with the greatest average slack is chosen. If no insertion points are possible then the route is abandoned.

This phase maintains a connectivity request matrix. It is updated for the best route that has been selected. A connectivity request is made between each node on the route (starting with the producer node) and the consumer node. The weight added is reduced by a percentage for each step after the producer. This ensures that the connectivity requests that have the greatest impact on the route length have the highest connectivity weight. The connectivity matrix has an axis with all operands and an axis all result ports. The weightings are modified by the total weight of the region being optimised. After applying to all regions in the application the matrix is used for allocating new connections.

During normal connection growth the highest weighted connection that can be added (as restricted by maximum connectivity length and maximum operand selections/output registers) is added.

10.2 Optimisation Principles

The output from the code translation process is an unoptimised CDFG. All register reads and writes in the host architecture are translated into register file read and write operations in the CDFG.

Transport optimisation is applied before generating the main code. The goal of the optimisation process is to remove unnecessary operations and dependencies between operations, in order to improve scheduling freedom. Primarily, the transport optimisation process seeks to remove many of the register file accesses. If data is written to a register and subsequently read by a later operation then, in many cases, the CDFG can be rewritten so that data is passed directly from one operation to the next. If a register is not live at the end of a strand then in many cases it is possible to completely eliminate the register write. These optimisations reduce the amount of bandwidth required to the register file (a considerable issue in most traditional processor designs) and to make use of direct connectivity between execution units. Using such direct connectivity can significantly enhance performance.

This optimisation process can, in a sense, be viewed as the implementation of the front end of a high end microprocessor in software. High end processors are able to perform dynamic instruction re-ordering and register renaming. Unfortunately, these facilities come at a considerable cost in terms of area, power and design complexity. The MetaMapper statically analyses code and reorders operations in the most efficient manner. Many accesses to the register file are optimised away to use direct paths between execution units, equivalent to the complex network of feed-forward buses in a high end processor. The hardware of a CriticalBlue processor remains simple and is controlled directly from a closely coupled execution word with the minimum of decode overhead.

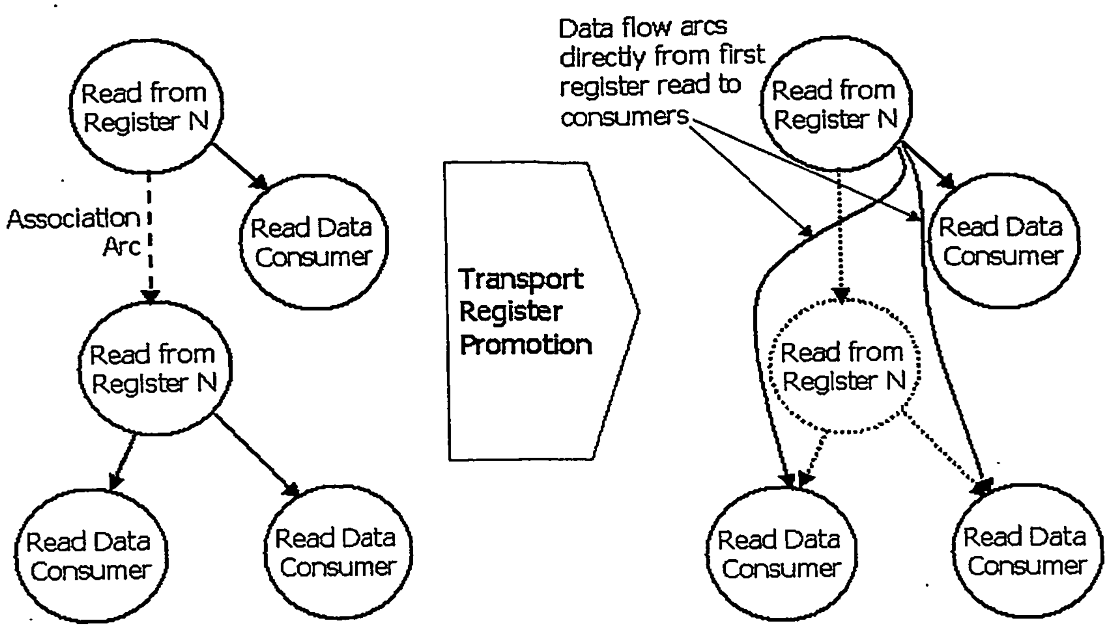
The CDFG optimisation process elucidates the data flows between functional units in the architecture. These data flows are then used during the architectural optimisation process to direct the connectivity between the functional units. If a particular data flow appears commonly, or in a particularly critical block of code, then this will in all likelihood lead the architectural optimiser to create a connection bus that corresponds to the data flow.

10.3 Register Promotion

Register promotion is an important optimisation that helps to reduce register file bandwidth pressure. In many cases a particular register may be read several times while holding the same value. This corresponds to the host code using the same register operand a number of times. If there are no intervening writes to the same register then all the reads will obtain the same value.

The purpose of the optimisation is to reduce the number of register reads so that only a single read is performed. The data obtained from the read may then be passed to all the operations that use the value. The data can be transported to the required operands over the connectivity network.

The diagram below illustrates the register promotion optimisation:



The left hand side shows a segment of the CDFG prior to the optimisation. There are two read operations from the same register. The first read has a single consumer of the data and the second read has two consumers. An association arc links the two reads. This generated when the CDFG was built and indicates that they obtain the same data.

The right hand side shows the CDFG segment after optimisation. The second read operation has been deleted as it is redundant. The data obtained from the first read is routed to the consumers of the second read. The dotted arcs have also been deleted.

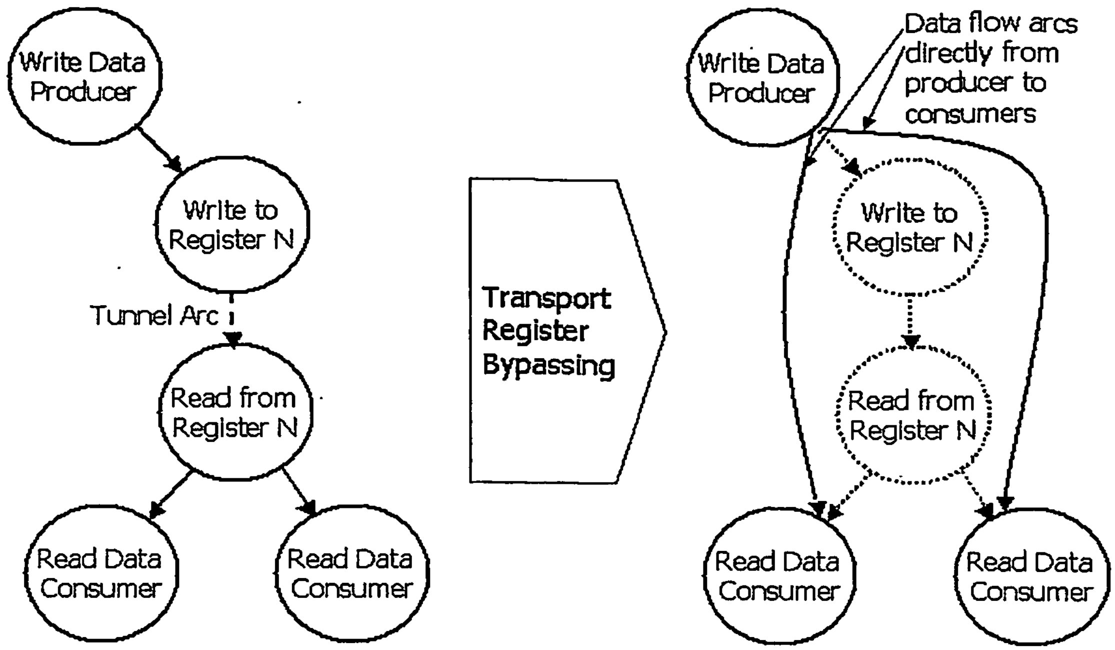
In general this optimisation can only occur if the reads are all in the same strand. However, in some circumstances the optimisation can be applied to accesses from different strands. The first read must be in a strand that is an atomic pre-dominator of the subsequent reading strands. That is, if the first read is executed then all subsequent reads are performed during the same execution of the region.

10.4 Register Bypassing

Register bypassing is another important optimisation that allows the elimination of both a register write and a subsequent read. The register promotion and register bypassing optimisations can be applied to the same segment of the CDFG, forming synergistic optimisations.

The optimisation occurs when a particular data item is written to a register and then subsequently read back within the same strand. The CDFG is rewritten so that the data passes directly from the data producer to the eventual consumers without having to pass through the register file at all. This optimisation can be on many occasions in typical code. Sequences that write to a register and subsequently read from it in the next instruction or within a few instructions in the same basic block are extremely common. In traditional processor architecture, reads of results in the next instruction would use a feed forward path around the register file. This optimisation represents a kind of software equivalent to this construct. Operation sequences are explicitly re-written to use direct scheduled paths through the connectivity network rather than the register file.

The diagram below illustrates a register bypassing optimisation:



The left hand side shows a segment of a CDFG before the optimisation. Data is calculated and then written to a particular register. The data then obtained by a register read and then passed to two consumer operations. A tunnel arc links the register write and register read. This indicates that data is "tunneling" through the register file and that the read definitely obtains the data stored by the write.

The right hand side shows the CDFG segment after optimisation. Both the register write and read are deleted. The original data producer passes its output to the data consumers, completely avoiding the register file.

In general this optimisation can only occur if the write and read are in the same strand. However, in some circumstances the optimisation can be applied to accesses from different strands. The write must be in a strand that is an atomic pre-dominator of the subsequent reading strand. That is, if the writing strand is executed then the reading strand must also be executed during the same execution of the region.

In cases where the optimisation cannot be performed because the write and read are in different strands and the write is not an atomic pre-dominator the scheduler may be able to fuse register write and read operations to the same register. That reduces the write and read into a single operation.

If the register is live at the end the strand then the register write cannot be deleted. In that case the bypass to the original data producer can still occur by the register write operation remains.

This optimisation allows greater scheduling freedom since the data consumers can be scheduled as soon as the data is available and the write can be scheduled later as it does not impact the placement of the consumers.

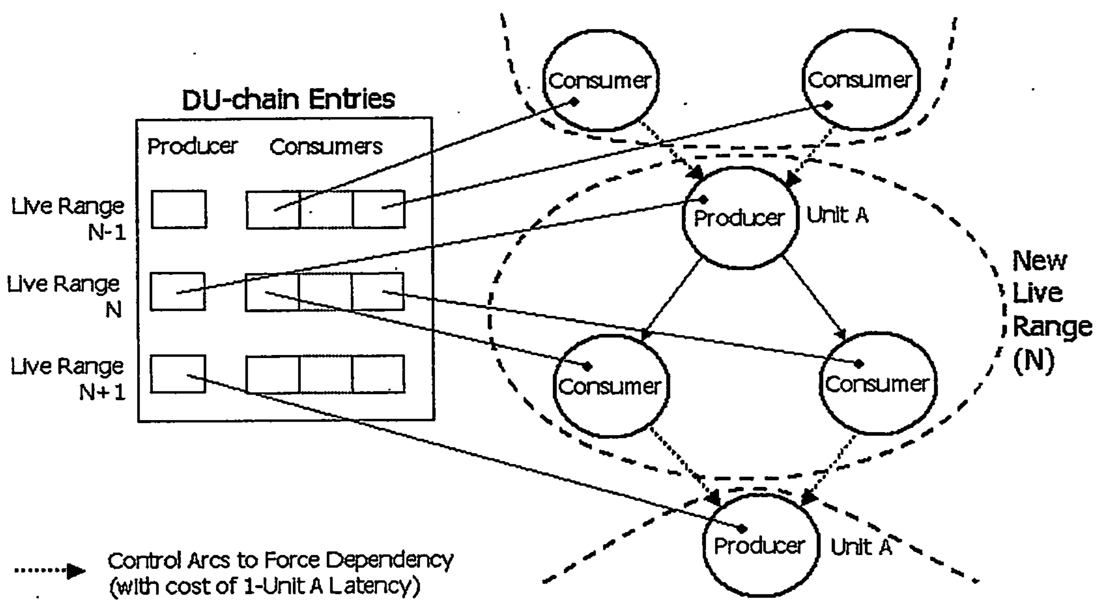
10.5 Live Range Insertion

Alongside the CDFG the DU-chain structure is maintained. As described previously this maintains the definition and use information for each output register within the architecture. The definition shows the node in the CDFG that generates a value in the

register and the use chain shows the nodes that consume that value. The point of the definition of the register to the issue of the last consumer of its value is its live range. Lives ranges for a particular register cannot overlap as all consumers for a particular value must be issued before a new definer can. If this rule is not observed then invalid results will be obtained.

As transport allocation is performed, particular live ranges may be deleted and new live ranges inserted. The appropriate arcs in the CDFG must be deleted as a live range is removed and new arcs added as a new live range is inserted.

The diagram below shows the duration of live ranges for a particular register with a new live range N being inserted into the CDFG.

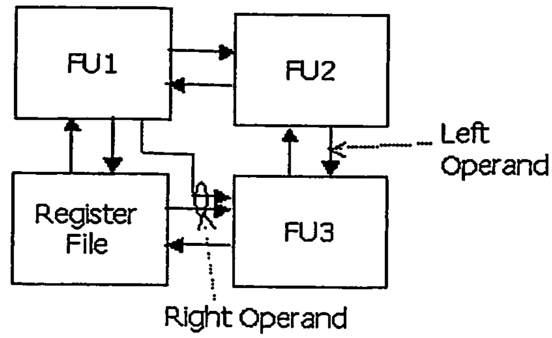


The consumers for the live range N-1 are shown. Dependencies are generated to the producer for live range N so that the register is not overwritten before all the consumers have read the data. The latency of the control arc is 1 – (the latency of the producer). Thus if the producer has a latency longer than 1 then the consumers might actually be issued after the producer. However, the dependency guarantees that the consumers will have read the data before the producer overwrites the register with a new value.

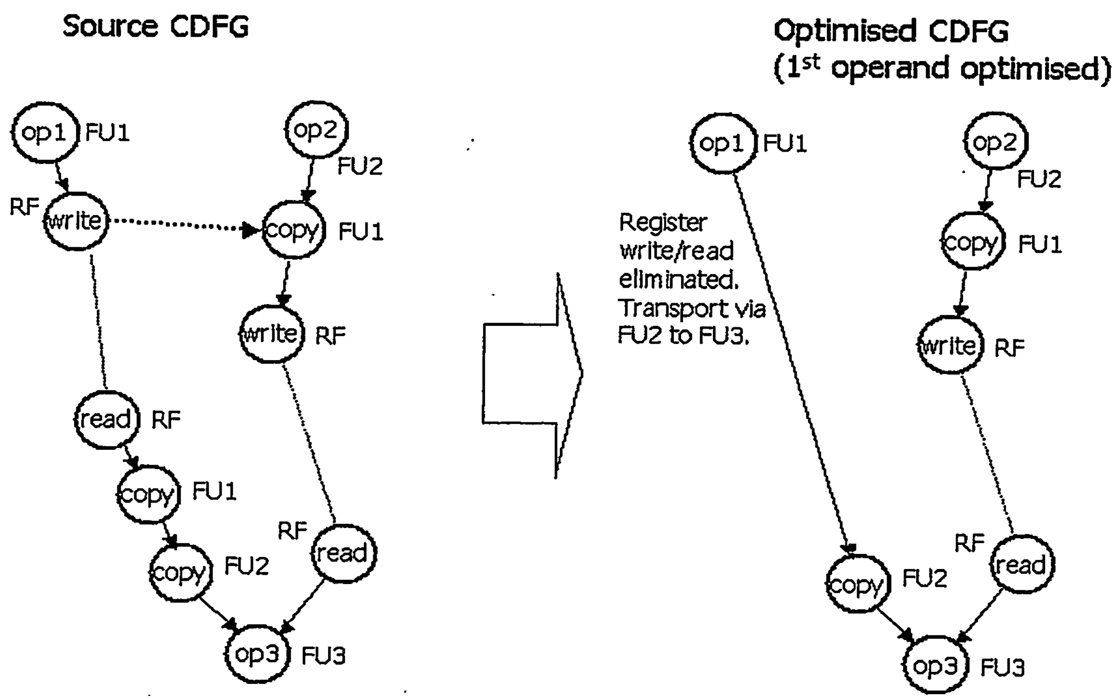
Data arcs connect the producer to the consumers. Finally the consumers of in the new live range have control arcs to the producer of the next live range.

10.6 Path Optimisation

The diagrams below show an example of path optimisation. The optimisation is based around the architecture shown below. As can be seen there is not full connectivity between all functional units so additional copy operations have to be inserted for certain data transports:

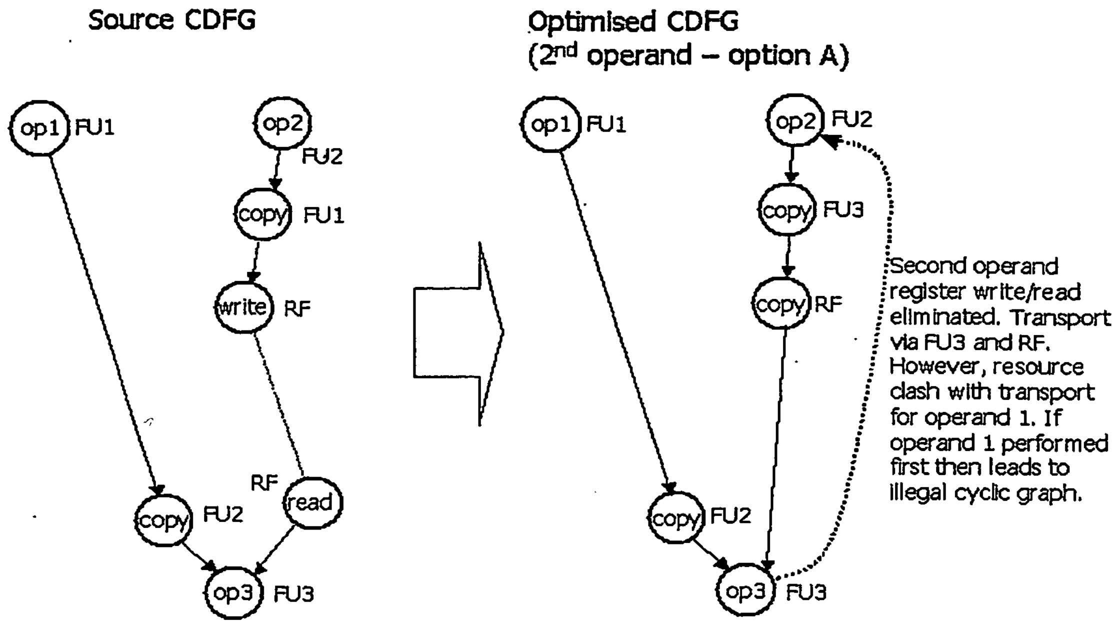


The following shows an initial CDFG on the left hand side and then an optimised CDFG on the right hand side. The example is identical to that used in the description of the initial transport allocation performed during the initial CDFG construction. Optimisations are performed in order of arc criticality so it is assumed that the arcs from op1 to op3 are more critical than those from op2 to op3. This is because there are more transport operations and thus greater latency in the former path. A register bypass operation is performed between op1 and op3. Since FU1 result (where op1 is mapped) and FU3 left operand (where op3 is mapped) are not directly connected a new copy operation is required. This is performed on FU2 to copy the result to the left operand.

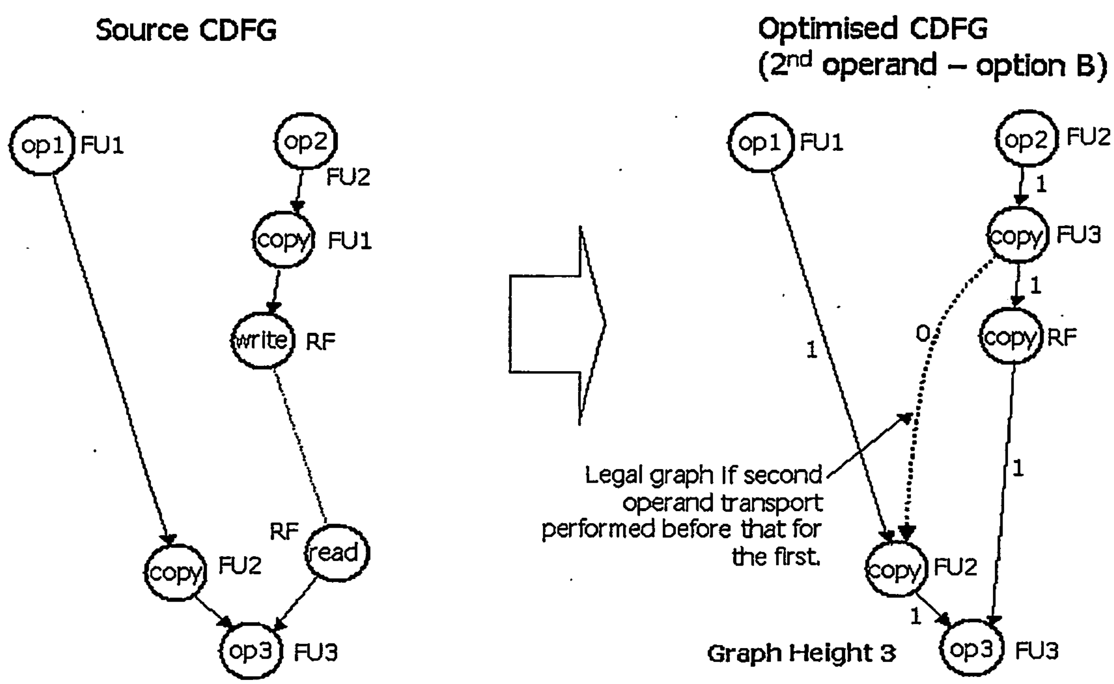


The next step is to try and optimise the transport between op2 and op3. Again a register bypass can be performed to eliminate the register write and subsequent read (it is assumed that the register is not live after the read usage). Since FU2 result (where op2 is mapped) and FU3 write operand (where op3 is mapped) are not directly connected then additional transport operations must be added. One possible route is via FU3 and RF and this is inserted into the CDFG. Thus the data is initially transported from FU2 to FU3. This is the same route that is being used to transport the other operand to op3. The live range insertion of the transport is after that for the transport for the left operand. Thus a dependency arc from op3 (the last consumer for the previous use of the register) to op2 is

added. However, this leads to a cycle in the graph. This is detected by updating the transitive closure of the graph. All graph additions that lead to a cycle graph are illegal and the particular transport optimisation is abandoned.

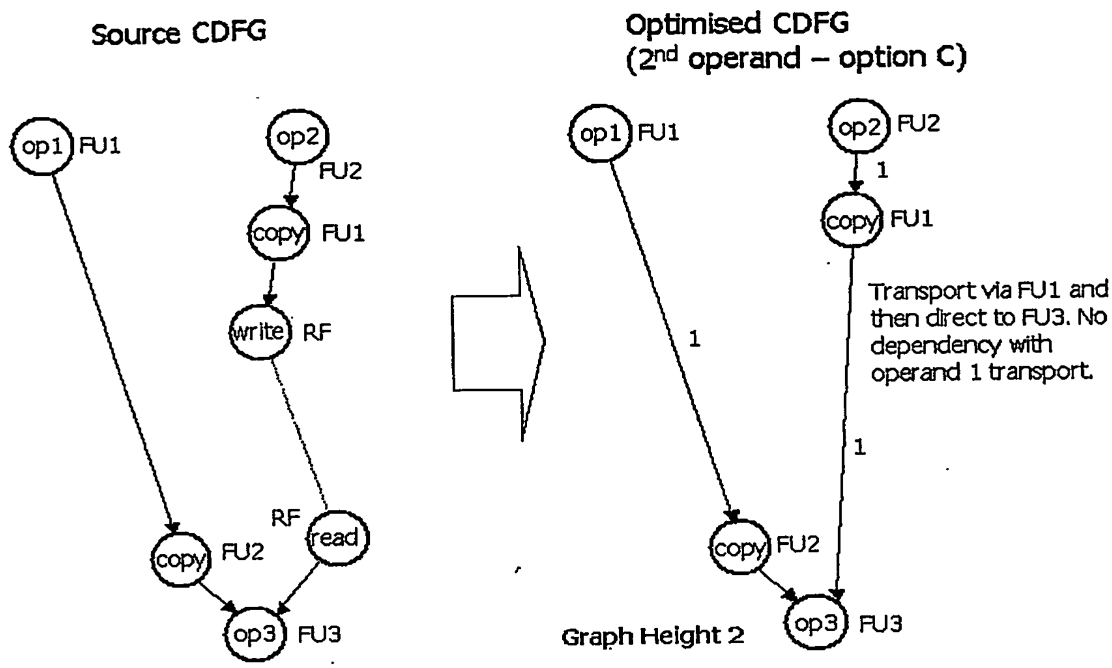


A further attempt is made at the same transport optimisation. In this case the live range insertion is performed before the usage for feeding the left operand of op3. In this case the optimisation maintains an acyclic graph and is legal. The new graph height is measured and found to be 3 clock cycles.



Finally a different transport route for the right operand of op3 is tried. In this case the data is transported counter-clockwise around the architecture to FU1 and then directly to the

right operand of FU3. Since this only requires a single copy operation it results in a graph height that is lower than that for the previous routing. Thus this is chosen in preference.

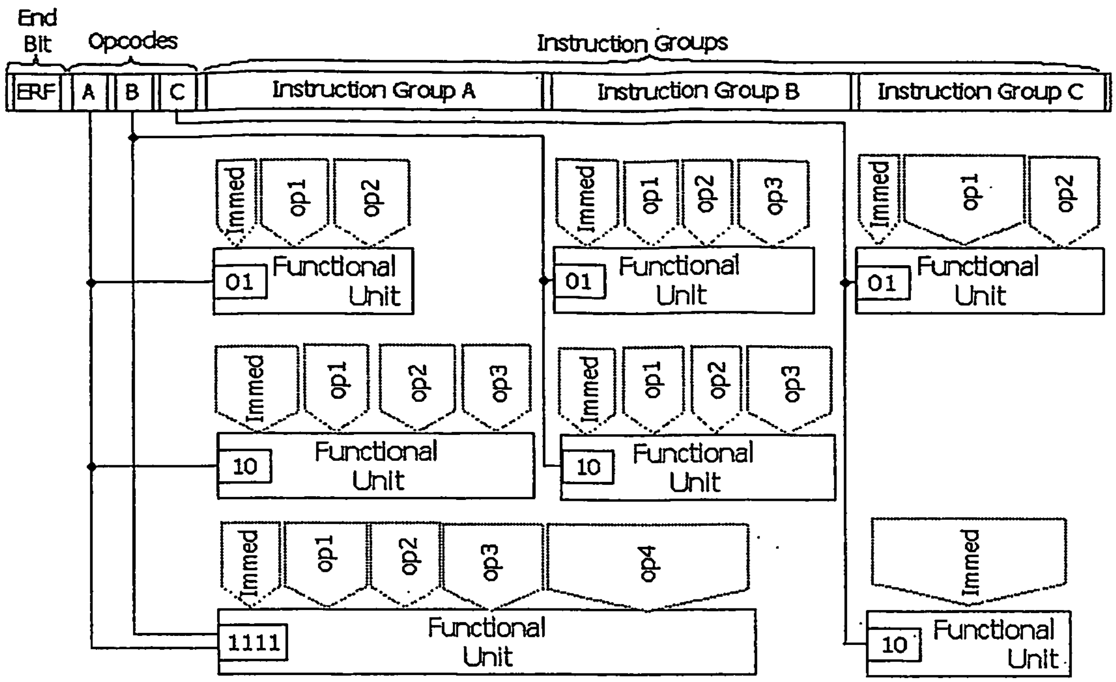


By choosing arcs for optimisation in order of their criticality, the most important data flows in the code are given the best choices of routes through the connections available in the architecture.

11 Execution Word Optimisation

11.1 Organisation

An example instruction word organization is shown in the diagram below:



The instruction word is divided into three sections, each occupying contiguous bits in the word:

- **End Bit:** This is a single bit used for specifying the end of the region. The bit is set for the last execution word in a region.
- **Opcodes:** This is a block of bits that are used to specify operation codes for enabling particular functional units. There are specific opcode bits for each group within the instruction groups section.
- **Instruction Groups:** This is the block of bits that actually control the individual functional units. The section is divided into a number of individual groups. The size of these groups is dependent upon the number of bits required to control particular functional units.

The diagram shows the required opcode bits to enable the use of a particular functional unit. This value is compared against the bits set in the opcode section. If there is a match then the functional unit is enabled. Only one functional unit from each instruction group may be enabled in each execution word. The opcode pattern 0 is reserved for each instruction group to specify a NOP (No Operation). If that pattern is used then no functional unit is enabled for the group.

The optimisation process determines the number of instruction groups and their widths automatically. In general, the most frequently used functional units are allocated into separate groups. This allows these units to be used simultaneously. Thus restrictions on parallelism due to layout interference between different functional units are minimised. Some functional units need a representation that uses more bits than can be specified in any one instruction group. In that case two or adjacent instruction groups may be used for the unit. The opcode sections for the groups are also combined and a unique opcode value is used from each individual group.

The number of bits required for each functional unit is dependent on a number of factors. Firstly, the method needs to be specified. The number of bits required is dependent upon

the number of individual methods for the unit. In some cases the method operand is also used for specifying immediate values. The remainder of bits are used to control the multiplexers for each operand. The number of bits required for each operand is dependent upon the number of sources that are selectable for the multiplexer.

Since the number of bits required for each individual functional unit differs, some bits may be unused within the instruction group depending on the unit selected. These unused bits are simply cleared.

Each functional unit only needs two contiguous groups of bits from the execution word to control it. Firstly, there is the opcode bus formed from one or more opcode sections in the execution word. Secondly, there is the instruction bus formed from one or more instruction groups in the execution word. This allows a simple specification of the connectivity required for a particular functional unit in structural HDL. A contiguous bus may also be easier for EDA tools to place and route efficiently.

11.2 Optimisation Process

Before valid code can be generated for a CriticalBlue processor, a layout for the execution word must be constructed. This allows the scheduler to determine which operations can be issued on the same schedule. Operations that share instruction groups cannot be simultaneously issued. The execution word optimisation process is executed each time a modified architecture is generated. As each new connection is added during architectural synthesis the execution word is re-optimised. This is because the addition of a new connection can increase the number of bits required to control a particular functional unit.

The placement of the control and opcode bits in the execution word for each functional unit are written out to the processor definition file during the synthesis process. This is file is read when generating code for the architecture so that the correct execution word layout can be generated.

11.3 Functional Unit List

A complete list of functional units in the architecture is created. The execution word must be constructed so that each of the functional units is controlled. A functional unit has bits allocated for control even if the unit is not actually utilised in the application code supplied to optimise the processor.

The list of functional units is obtained from the hardware configuration file. The required bit width is calculated for each functional unit. This is based on the width of the method port, any immediate and strand ports and the bits required for controlling the operand multiplexers. The number of bits required for the operands is dependent upon the number of data sources that need to be selected between. This changes as new connections are added during the optimisation process.

11.4 Functional Unit Frequency Calculation

In order to generate an optimised execution word information is required about the frequency of use of each of the functional units. This allows improved allocation within the execution word. The intent is to allocate the most frequently used functional units into different instruction groups in the execution word. This allows operations to these functional units to be issued simultaneously and thus reduce the number of incidents in which parallelism is limited by the placement of operations in the execution word. Functional units that are rarely used can share bits with more frequently used operations within the same instruction group.

The frequency of usage is based upon a static count of operation types within the CDFG. A dynamic count cannot be obtained since it is not known how many times particular loops will be executed. Instead, an arbitrary fixed multiplier is applied to operations within recognized loops. This increases the frequency of use of operations within the loop. The frequencies are also weighted by the criticality factor allocated to the critical functions within the application. Thus the frequency of operations within particularly critical functions is given more priority than operations in less performance critical control code. The frequencies are calculated by performing a partial scheduling pass. The CDFGs for all the code is generated but no actual scheduling is performed. This allows a traversal of the CDFGs to be made in order to determine the usage frequencies.

The frequency analysis is performed each time new connections are added to the architecture during synthesis. This is because the addition of new connections changes the conenctivity network within the processor for transporting data items. This in turn can change the amount of usage of a particular functional units for copying data items over that network. This needs to be reflected in the allocation of the control bits for the register.

A frequency per bit value is calculated for each functional unit. This is simply the weighted frequency for the unit divided by the number of bits it occupies. This value is used when allocating instruction groups within the execution word. A frequency density is used so that very wide but frequently used units are not allocated a complete instruction group within the instruction word. This is inefficient in terms of code density. It is better to allocate smaller instruction groups and then concatenate them as required to facilitate wider operations.

11.5 Layout Creation

The layout creation algorithm uses multiple passes in order to converge on a particular layout. Multiple passes are required since it is not possible to determine beforehand the number of bits that need to be allocated to opcode bits within the execution word. The number of bits required depends on how many operations are packed into each instruction group. Each operation must be allocated a unique opcode so the required width increases with the number of operations. The placement of functional units to particular instruction groups depends on their relative usage frequency.

An iterative scheme is used that initially allocates a total of 0 bits to the opcode section of the execution word. The placement of the units is performed and if the required opcode width exceeds the number of bits allocated then the process is restarted with one extra opcode bit. The process ends when the first complete allocation can be performed. If there are few functional units and a wide execution unit then it is possible that an allocation with 0 opcode bits could succeed, if each functional unit can be allocated to an independent instruction group.

11.6 Instruction Group Placement

The instruction group placement determines the number and width of each individual group within the execution word. The placement is performed on the basis of functional unit frequency per bit. The functional units with the highest such values create the allocation of an instruction group of the required width. This will later be allocated to the particular unit during the unit placement step.

Instruction groups are allocated in priority of frequency per bit of functional units. Thus the first instruction group is allocated on the basis of unit with the highest frequency per bit. The second group is allocated on the basis of the unit with the second highest frequency per bit and so on. This allocated continues until there are no bits remaining to allocate any further groups. The total number of bits that can be allocated is based on the total

execution word width minus the total opcode bits reserved and a bit for the end flag. If a particular allocation cannot be performed because the unit requires more bits than those remaining then the algorithm moves to the next unit in frequency per bit order.

A single bit is allocated for each group for opcode usage. Each group needs at least one opcode bit because it must support a no-operation with an opcode of 0. The allocated unit is given the opcode value of 1. If additional operations are to be supported by the group then further bits need to be allocated from the opcode pool.

At the end of the allocation there may be some bits remaining that cannot be usefully used for instruction groups. These additional bits are allocated to the total that are reserved for opcode usage.

11.7 Allocation of Units to Groups

During this phase each functional unit is allocated to the execution word. The list of functional units is processed in their absolute frequency order. Thus the functional unit with the highest frequency is allocated first, then the unit with the next highest usage and so on until all units are allocated. A cumulative total of the frequency of all units allocated to a particular instruction group is maintained. The aim of the algorithm is to balance the allocations so that all instruction groups are utilised equally as much and thus highly used units are allocated to different groups. The allocations are performed in frequency order because the allocation of a unit to a particular group affects what other units are allocated to that group. The most frequently used units are allocated first so that they are given the greatest freedom of choice.

A particular unit may require more than one instruction group if the number of bits it requires is greater than available in any single group. Alternatively, a unit may be allocated to multiple groups, even if there is a single group wide enough, if the usage frequency of the wide group is very high based on previous placements. If a unit is allocated across multiple groups then they are always contiguous groups. This ensures that the opcode and control bits for the unit are always composed of contiguous bits from the execution word.

The algorithm creates a list of candidate placements for the unit. Placement is tested with each of the instruction groups as the leftmost group. If that single group is wide enough then it is added to the candidate list. If not then additional groups are included until the set of groups are wide enough. The set of groups is added as a candidate. If the set cannot be made wide enough then no candidate is added.

A check is made of the total opcode width that needs to be decoded for the unit. This is formed from the same of the opcode widths for each group that the unit occupies. If a unit is allocated across multiple groups then the opcode width can expand significantly. There is a maximum opcode width imposed by the number of bits that are decoded by the controller for the unit. If that width is exceeded then the placement is not added to the candidate list.

A calculation is also performed to determine how many additional opcode bits will be required if the placement is used. Additional opcode bits may have to be allocated to each group if the full set of values is already used. A unique opcode must be allocated for each group that the candidate placement occupies. A bit expansion value is calculated for each candidate that indicates the number of additional bits from the total opcode pool that the placement will require.

Once a candidate list has been generated the algorithm selects the best placement. If the candidate list is empty then that indicates that a placement could not be made. This can

only occur if the number of bits required for a particular unit exceeds the bits available in the execution word, excluding the opcode and end bit. In that case a wider execution word must be used.

The candidate is chosen from the list on the basis of the minimum bit expansion with secondary criteria of the minimum total frequency. Thus placements tend to be chosen that do not require many additional opcode bits. The total frequency is the sum of the frequencies of all units allocated to the instruction group including the new placement. By using the placement with minimum frequency we minimise the chances of clashes between the units during operation scheduling.

If additional opcode bits are required then the allocation process must be restarted with an extra opcode bit. This reduces the number of bits that can be allocated for functional unit control and may have an impact on the number and size of the instruction groups.

12 Operation Scheduling

12.1 Overview

The CriticalBlue tools generate code for the architecture using advanced global scheduling techniques. Code generation and scheduling for the CriticalBlue architecture are more difficult than for a traditional architecture. This is because the connectivity between the functional units is sparse and non-symmetric. The placement of particular operations onto particular functional units must be cogniscent of the difficulty of supplying operands to the unit.

The output from the code translation stage of the tools is a CDFG (Control and Data Flow Graphs) that represents the data flow between operations. A separate CDFG is generated for each individual region in the code. An individual region may contain control flow as well as data flow constructs. The CDFG representation maps control operations to predicate operations so that they may be represented in the graph.

The original register allocation of the source code is maintained. However, the register file is treated as an ordinary functional unit and reads and writes to the register file have to be explicitly scheduled as separate operations. The overall register file may be split into a number of separate units that partition the registers in a fixed manner. Each such functional unit as a single access port so the overall bandwidth to the register files is considerably lower than for a traditional architecture. This avoids much of the scalability issues normally associated with the register file.

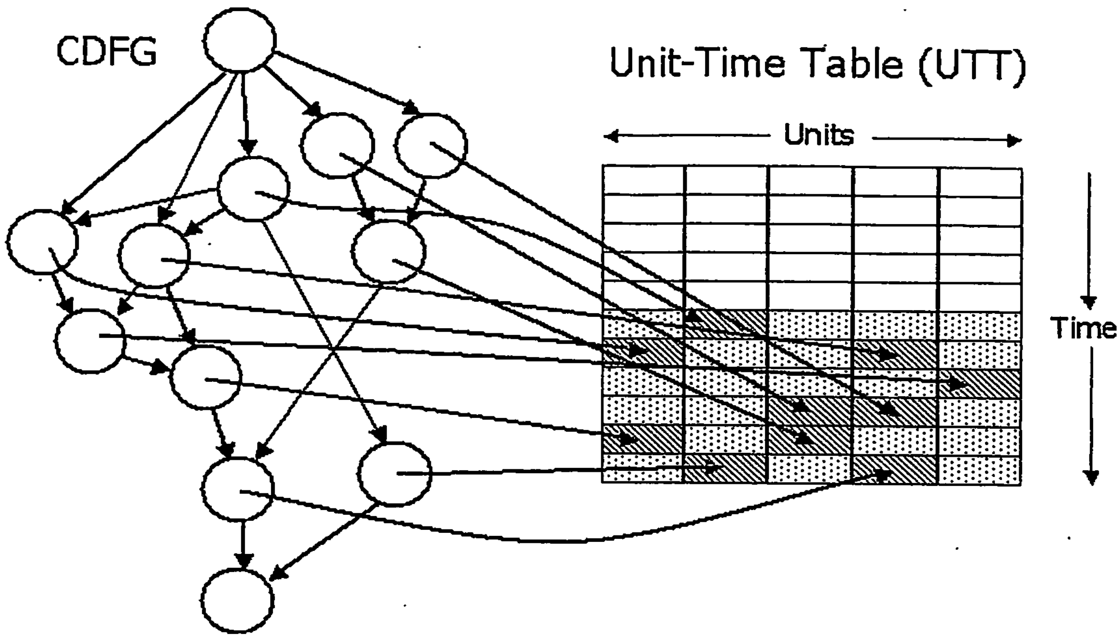
Communication between the register file and the other functional units is via explicit operations to move the data directly to the required functional unit or through intermediate functional units operating in a copying capacity. The processor connectivity is optimised so that most critical operations can be performed by transferring data between functional units. Thus access to the register file is not required. This significantly reduces the bandwidth requirements of the register file.

12.2 Unit-Time Table

The scheduling problem becomes one of mapping the CDFG onto the processor bus connectivity. The CDFG represents the static flow of data in the code. A unit versus time representation is used to show how the available bus connections are used to transport data around the network. In general each operand can select data from one of a number of input buses. The selection can be changed from one clock cycle to the next to implement the data flow of the CDFG. The unit-time representation and these selections

ultimately represent the sequence of code in the region. Once the selections for each cycle have been established it is a simple matter to translate that into a binary machine code representation.

The diagram below shows an example mapping from the CDFG to the Unit-Time Table (UTT):



The lines from the nodes to the graph represent the allocation of a particular operation to a particular unit on a particular clock cycle in the schedule. The unit type is dictated by the operation being performed. In some cases there may be multiple execution units of the same type in the architecture, providing a choice about which particular unit a given operation should be mapped to. The scheduling starts with the last operations from the CDFG and gradually works up the CDFG to the earlier operations. Thus the unit-time table is filled from the bottom upwards and expanded as required to provide sufficient clock cycles to execute the operations in the CDFG.

An operation is only scheduled when all its consumers have already been scheduled in the unit-time table. Thus a bottom up algorithm is used that schedules the root operations first. These are operations that have no consumers in the same region. Thus code is effectively scheduled backwards.

12.3 Node Criticality Analysis

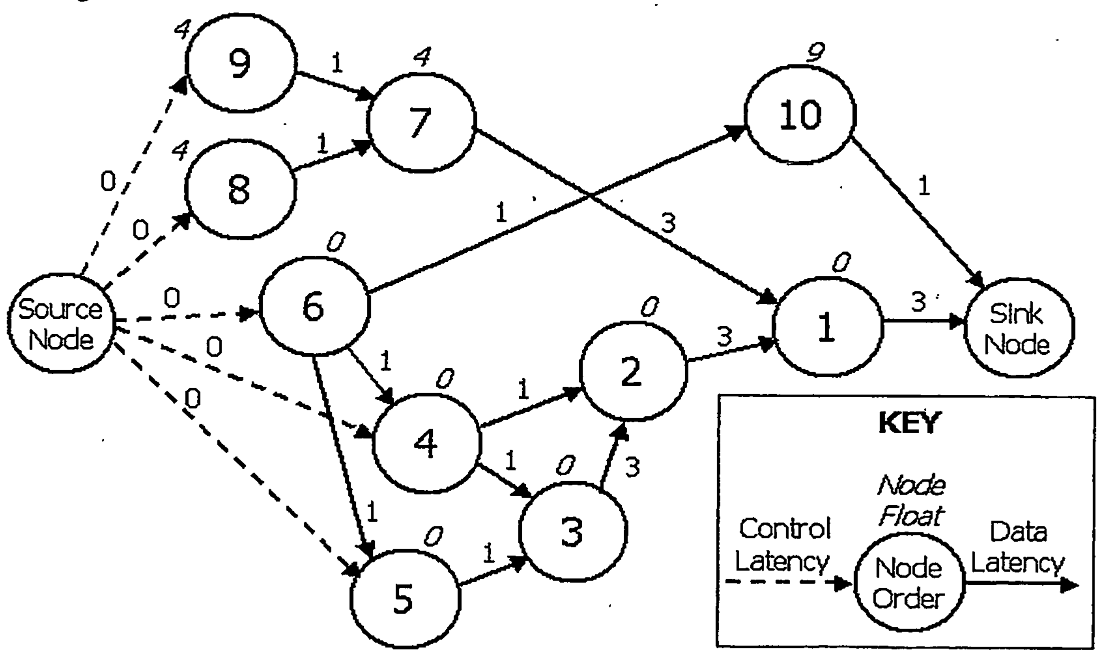
Critical path analysis is applied to the CDFG to determine the most performance critical data flows through the graph. Each node is assigned a free float value by the critical path analysis. This is a measure of the number of clock cycles of freedom the operation has in terms of its placement in the schedule before it starts to impact the overall execution time. If a node has a free float of 0 then that indicates that it is on the critical path through the CDFG.

The nodes within the CDFG are prioritised depending upon their free float values. The nodes with the lowest free floats are given higher priorities. Many nodes will have the same free float value. In that case nodes with the greatest number of inflow arcs are given

a higher priority. This is because these nodes are greater consumers of earlier nodes and thus their placement is likely to allow a larger number of producer nodes to be subsequently scheduled. Thus their earlier placement allows a greater amount of scheduling freedom. If nodes have both the same free float and inflow arc count then the prioritisation is determined by their position in the original code. Nodes associated with later host instructions are given higher priorities. This helps preserve the original ordering of operations in the absence of any other constraints and thus reduces excess register pressure.

The most critical operations are always scheduled first. These operations, whose placement can have the most impact on the overall execution time for a block of code, are given the earliest choice of data transports between functional units. Less critical operations may be delayed longer, waiting for suitable data transports between functional units.

The diagram below shows an example CDFG with arc latencies, node floats and node ordering:



Each arc is labeled with a cost. This corresponds to the latency of the operation associated with the node. Nodes on paths with long latency operations tend to have higher criticality.

The figures by the nodes themselves are the free float values. These are calculated as by the critical path analysis. The figures within the nodes are the node orderings. The lowest numbers are the most critical nodes.

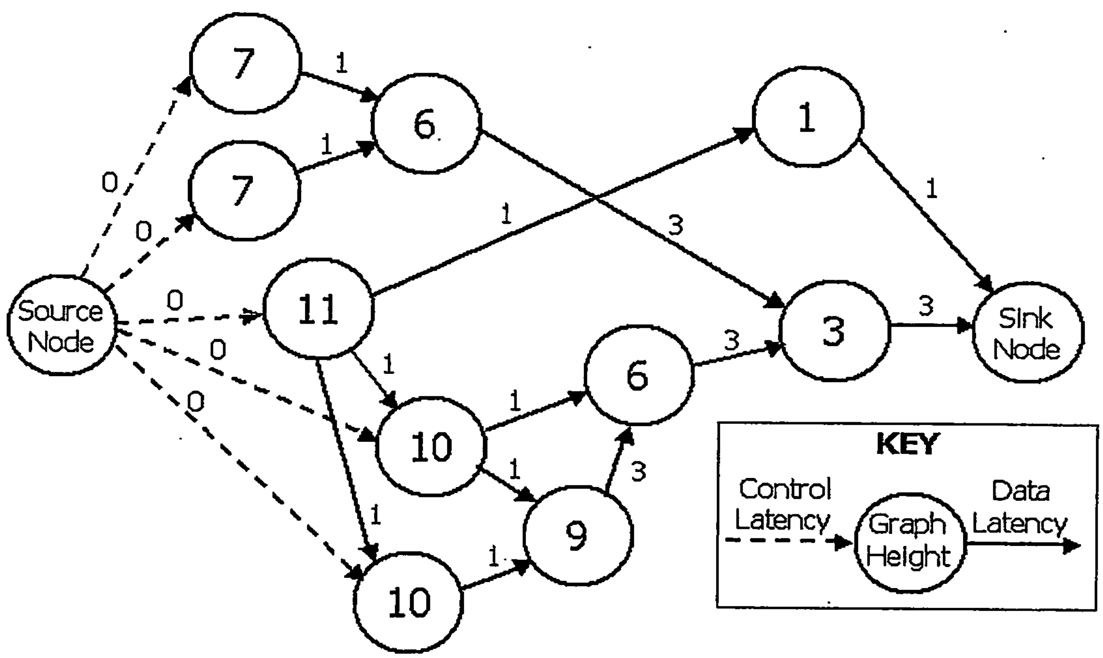
The node ordering represents the order in which nodes will be selected for placement during the scheduling process. Only nodes whose children have already been scheduled will be selected for scheduling. However, within the set the selection order is based on the node ordering value.

The critical path analysis includes the weak arcs when analyzing the CDFG. This is because the scheduler attempts to generate a schedule that does not violate the weak arcs. Conditional and association arcs are ignored during the critical path analysis.

12.4 Graph Height Analysis

In addition to critical path analysis, graph height analysis is also performed. The result of this is a height value for each operation node in the CDFG. This represents the minimum number of clock cycles from the initiation of the operation to the completion of the region represented by the CDFG. In effect the tree height is a measure of the schedule length on an ideal processor where they are no limitations on the placement of operations in the schedule other than the dependencies between them and the latencies of the operations. In reality there are other constraints that result in longer schedules than suggested by the tree height values.

The diagram below shows an example of a CDFG and the tree height calculation:



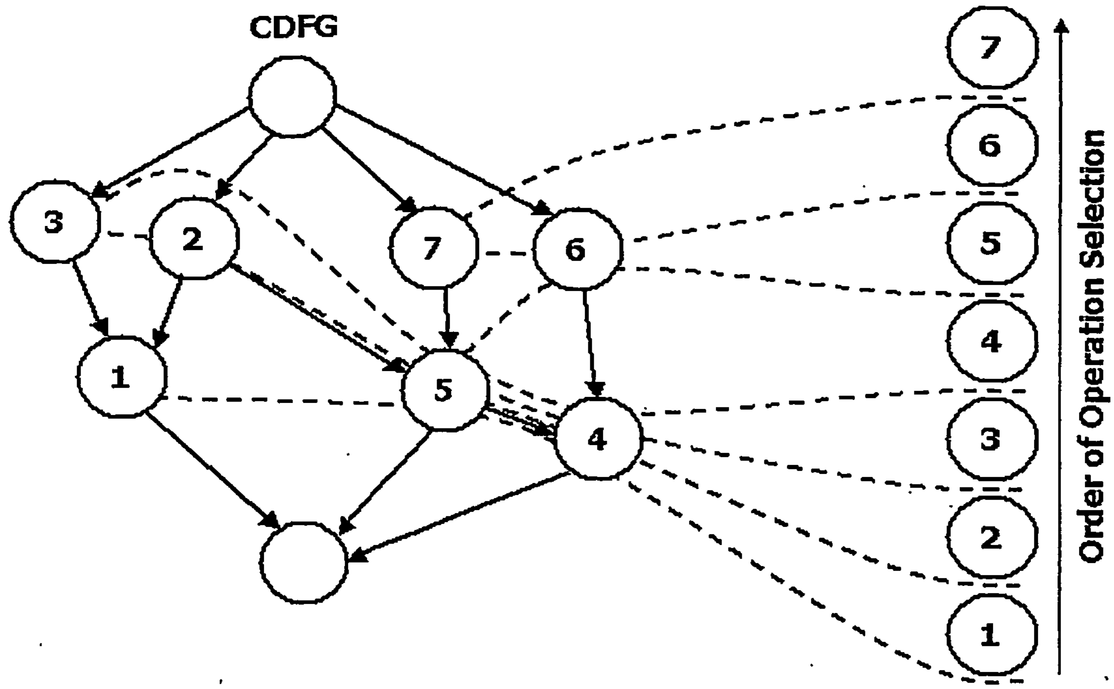
The latencies for each arc are shown and the tree height is shown inside each node. The height is determined from the maximum of the length of an outflow arc plus the height of the child node.

The tree height is used when determining whether to violate weak arcs when generating a schedule. The placement position of an operation may depend on whether weak arc dependencies are violated or not. The difference in potential placement position is calculated as a percentage of the tree height for the node to determine whether to violate the weak arc dependency.

12.5 CDFG Traversal

To generate a schedule all nodes in the CDFG must be traversed. As each node is traversed the operation associated with the node is placed into the Unit-Time Table. The traversal is via a depth first ordering mechanism. That is, a node is only scheduled when all the children of that node have already been scheduled. The CDFG is acyclic so there are no potential circular dependencies that could cause a deadlock. The sink node is the last node in the CDFG and this node does not have to be issued. The initial set of nodes that can be issued are those that only have outflow arcs to the sink node.

The diagram below shows an example of a CDFG traversal:



The CDFG is shown on the left and the sequence that the nodes are scheduled is shown on the right hand side. The first node to be scheduled is shown at the bottom of the diagram. Note that the order of operation selection does not necessarily correspond to the temporal order of operations in the Unit-Time Table.

At the start of the scheduling process, the nodes 1, 4 or 5 could be issued as they only have outflows to the sink node. The dashed line shows the set of nodes that can be selected at each stage. The choice of node will be based on the previous calculated node criticality. In this example node 1 is chosen. The choice of nodes is then opened to include the parents of node 1, nodes 2 and 3. The dashed line shows that these are included in the selectable set. This scheduling process continues until all operation nodes from the CDFG have been scheduled. The source node for the CDFG does not need to be issued.

An operation node must be issued before all its child nodes in the schedule but another independent node could actually be issued after it in the schedule even though it was selected from the CDFG later. For instance, in the example node 4 might be scheduled after node 3 if node 3 represents a high latency operation that takes a number of clock cycles before it can feed data to node 1.

12.6 Individual Operation Scheduling

The CDFG traversal mechanism determines the order in which operations are selected for placement into the Unit-Time Table. This section describes how an individual operation is scheduled into the table.

12.6.1 Latest Start Time Determination

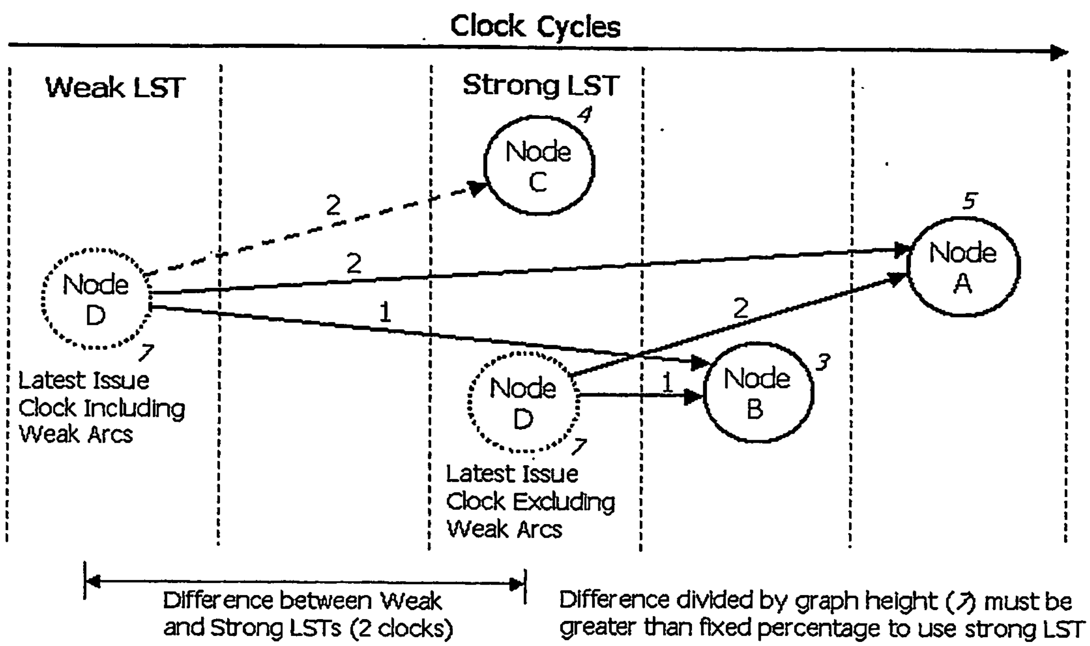
When an operation has been selected for placement the first requirement is to determine its latest start time. This is the last clock cycle and which the operation could be issued and is able to feed data results to consumer nodes. The latest start time also takes control arcs into account to ensure that an operation is issued a suitable number of cycles before any control dependent operations.

The Latest Start Time represents the very last cycle on which the operation could be issued. There may be other constraints, such as the availability of issue slots and extra time required to transport data items that may mean an operation has to be issued in an earlier cycle. In general the scheduler tries to issue each operation as late as possible as they will tend to result in the shortest overall schedule length (and thus best execution time).

An operation is only selected for scheduling once all child nodes have already been scheduled. Thus the clock cycle in which they are placed in the Unit-Time Table (UTT) is known.

If a node has outflow weak arcs then two different Latest Start Times (LSTs) are possible. One takes account of all arcs and the other ignores weak arcs. The variant that ignores the weak arcs may allow a later LST.

The diagram below shows an example of the LST calculation:



A small segment of a CDFG is shown. Node D has been selected for issue. It has data arcs to nodes A and B that have already been issued. Looking at the latencies of those arcs gives the strong LST. However, the node also has a weak dependency arc to node C. If this is taken into account then this gives the weak LST. In this example there is a difference of two clock cycles between the two.

The scheduler must select which of two possible LSTs it will select. In general it is trying to choose the latest start time possible. However, if the operation is issued after the weak LST then the weak arc dependency has been violated. If this is the case then associated conditional arcs are enabled, meaning that extra operations will have to be added into the CDFG. Moreover, this also means that the region will have to be dynamically re-executed. This lowers overall performance. Thus a weak arc is only violated if there is significant difference in the two LST values. The "significance" of the difference is measured as a percentage of the tree height of the node. Thus a greater difference is required to violate a weak arc if the node is part of a long chain of calculations and thus has a large tree height. In the example the difference between LSTs is divided by the tree height of node D that is 7 (the tree heights are shown by the nodes on the diagram). If the value of 2/7 is

greater than a fixed threshold then the weak arcs may be violated and the strong LST selected.

If code for a critical function is being scheduled then weak arcs are never violated and thus the weak LST is always used. This is because performance is considered to be more critical than code density in such functions.

As nodes are issued the cycle in which they were eventually placed is stored as an attribute of the node. This allows the LSTs for parent nodes to be easily calculated when they are selected for issue. The placement positions are kept as negative clock numbers measured relative to the last cycle of the UTT. This avoids the requirement to renumber them as new cycles are added to the start of the UTT.

Nodes at the end of the CDFG only have outflows to the sink node. The latency of the arcs is 0. The sink node marked as being issued on the very last cycle of the UTT. Thus the LST for such a node is also set to the very last cycle of the UTT.

12.6.2 Earliest Start Time Determination

In addition to the LST, an Earliest Start Time (EST) is also calculated. The combination of the EST and LST provide a window range in which an operation may be issued. The EST is fixed as being a certain number of clock cycles prior to the LST. The number of clock cycles is a fixed proportion of the total CDFG tree height (i.e. the tree height of the source node). Thus operations in a large CDFG are given a wider window than those in a smaller CDFG. This is because variability in the issue of a single operation has a smaller percentage impact on the overall execution time.

12.6.3 Operation Placement

Once the EST and LST have been calculated the operation is placed in the UTT. Initially an attempt is made to place the operation in the LST cycle. If this is not successful then an attempt is made to place it in the cycle before. This continues until the EST is reached. The cycle that is selected may be earlier than the first cycle in the UTT. In that case the UTT is extended with additional cycles at the start of the table.

If the operation is eventually placed after the weak LST time then the weak arcs have been violated. The appropriate conditional arcs associated with the node are marked as being activated. This causes the conditional nodes to be issued as active operations. Note that if the strong LST is selected the earliest possible issue time for the node may in the weak LST or earlier. Thus the weak arcs may not actually be violated.

The layout of the execution word may share certain bits between multiple functional units. Thus operations cannot be simultaneously issued to units from the same instruction group. If an operation has been issued on a cycle to another unit in the same group then that precludes issues to other units in the same group on the same cycle.

12.6.4 Inactive Conditional Nodes

If an inactive conditional node is encountered during scheduling then it can be scheduled without having to allocate any resources within the UTT. The nodes are simply marked as being issued in their LST cycle. The issue of the node may allow additional nodes to be added to the issueability set. A conditional node is inactive by default and only becomes active if there has been a dependence ordering violation of an associated weak arc.

12.6.5 Register Write/Read Merging

In some circumstances it is possible to merge write and read operations of the same register in a single operation. The register file supports three methods. There are read

and write methods (that appears as operations in the CDFG) but there is also a combined write/read. This operation is only issued when a merging opportunity is encountered. The merged operation performs a write but simultaneously forwards the data to the result port of the unit. The main advantage of the operation is that the write and the read can be from different strands. If the writing strand is disabled then the merged operation becomes a read operation on the register file.

Whenever a write operation is being issued a check is made to see if the earliest dependency is a read from the same register. If so then the read operation in the UTT is modified into a merged write/read. The write is then marked as being issued on the same cycle as the read.

13 Code Generation Algorithms

13.1 Executable Image Handling

13.1.1 Top Level

```
func_list = CreateFunctionList(used_map, fixup_list)
func_addrs = empty

// obtain the architecture on which to operate
if in synthesis mode then
   // if we are in synthesis mode then we need to generate the base
   // architecture including the default connectivity
   generate the base architecture
else
   // if we are generating code then we read the current architecture and
   // copy the current executable image to the shadow section
   read the current architecture from the description file
   copy the current executable image to the shadow section of the
      executable (it is preserved to allow the monitor to return
      values from it rather than the actual code)
endif

// If in code generation mode then only one pass at scheduling the code is
// made. If in synthesis mode then a number of passes are made until no
// more useful resources can be added to the architecture within the area
// constraints. The first pass is a dummy pass used to gather operation
// frequency information to determine the instruction layout
generation = 1
do
   if in synthesis mode then
      create empty resource request for each critical function and one for
         all other functions
      valid_layout = false
      if generation > 1 then
         // an instruction layout is generated for all but the first
         // generation of the architecture - there is no operation
         // frequency information to produce a layout for the first pass
         valid_layout = GenInstructionLayout(instruction width)
      endif
      if not valid_layout then
         clear the instruction layout
      endif
   endif

   // functions visited in criticality order for synthesis to
   // optimise instruction layout around most critical functions
   func_addrs = empty
   foreach function in func_list in criticality order do
```

```
// if we are just generating code then check to see if the code
// has actually changed from the database version - if not then
// we do not need to reschedule it
sch_func = true
if in code generation mode and function is in code database and
        old and new code are identical (ignorning const data
        and external labels) then
   // the code itself has not changed but we may still have to
   // regenerate it if it calls a function whose inflow liveness
   // has changed (i.e. it uses different parameters). In general
   // the calling code will have changed to accomodate the change
   // in parameters - this is really to handle any case where
   // there is no type safe linkage
   force_sch = false
   foreach function called by the code do
      if function has different liveness in current code then
         force_sch = true
      endif
   endfor
   if not force_sch then
      sch_func = false
   endif
endif

// go ahead and generate the code image for the function - either
// from the existing code database or a new schedule for the code
if sch_func then
   // generate new code for the function
   func_type = Normal
   if function is in fast interrupt list of config file then
      func_type = FastInterrupt
   else if function is in slow interrupt list of config file then
      func_type = SlowInterrupt
   else if function is in the SWI handler list of config file then
      func_type = SWIHandler
   else if function is in the critical list of config file then
      func_type = Critical
   endif
   UTT_list = ScheduleFunction(start address of func,
              end address of func, func_type,
              func_list, connection requests for function)
   if in synthesis mode and func_type = Critical then
      // for critical functions in synthesis mode we estimate the
      // execution time by adding together the lengths of the deepest
      // loops
      length = 0
      foreach UTT in UTT_list at deepest loop level do
         length = length + UTT schedule length
      endfor
      write out details for function length estimate
      examine UTT_list and work out usage percentage for each
           functional unit - write this out
      add length to total execution time metric (weighted by function
           criticality)
   else if in code generation mode then
      // generate the machine code for the function from the
      // scheduled Unit Time Tables (UTT)
      code_blocks = BuildBinary(UTT_list, block_fixups)
      if code_blocks only calls named functions (or indirect) then
         write code_blocks to code database (main and shadow)
         write list of named called functions to database along with
              their inflow liveness
         write block_fixups to code database
      endif
   endif
else
   // extract the existing version of the code from the database
   code_blocks = existing version of code from database (main and
```

```
                 shadow)
        block_fixups = fixups associated with the code block from the
                 database
      endif

      // if generating code then the image for the function needs to
      // be written to the final executable
      if in code generation mode then
         WriteBinary(code_block, used_map, block_fixups, fixups_req)
         add function start address to func_addrs
      endif
   endfor

   // if we are synthesis mode then we need to write out the details of
   // the current architecture (estimated execution time, area, code size)
   // and determine whether to add a new connection to the architecture
   resource_added = false
   if in synthesis mode then
      write out details of current generation architecture (including area,
         total execution time metric and code size)
      write out information about percentage usage of each functional unit
         (weighted as required) - this may be used to determine if new
         functional units need to be added by the user
      generation = generation + 1
      weight connection requests by weight of individual functions (with
           all other functions grouped together)
      resource_added = add a new connection from cumulative connection
                    requests
   endif

while resource_added repeat

// remove dead entries from the code database, resolve all the fixups
// encountered within the code and write the function address table
if in code generation mode then
   remove any entries in the code database that were unused
   foreach item in fixups_req do
      modify the code image with the fixed up address from func_addrs
   endfor
   write the func_addrs to the shadow section - this is used to map from an
      original PC value to the translated code (single entry for each
      function that covers the PC range to the next function)
endif
```

13.1.2 Build Binary

```
CodeBlockList BuildBinary(UTTList, BlockFixups)
// Takes the list of Unit Time Tables for regions within a function
// and builds a binary image. As branches/calls in the translated code
// are encountered they are added to the BlockFixups list with a blank
// address written to the code block. Returns a list of code blocks
// that form the function.
// UTTList is a list of Unit Time Tables (UTT) for regions in the
//    function
// BlockFixups are the list of fixups required for the code blocks
```

13.1.3 Write Binary

```
void WriteBinary(CodeBlockList, UsedMap, BlockFixups, FixupsReq)
// Writes a list of code blocks to the executable image. The code blocks
// may have been retrieved from the code database or have just been
// scheduled. The code blocks are placed into unused words within the
// executable image. The UsedMap shows which locations have been used or
```

```
// hold data or return links etc. As fixups in the BlockFixups are
// encountered and the exact location becomes known then they are added
// to the full FixupsReq list.
// CodeBlockList is a list of code blocks to be written
// UsedMap is a bitmap of used locations in the executable image
// BlockFixups is the list of fixups with the code blocks
// FixupsReq is the full list of fixups and is appended as required
//    by fixups within the code blocks
```

13.2 Function Handling

13.2.1 Function List Creation

```
FunctionList CreateFunctionList(UsedMap, FixupList)
// Examines all the code in the executable and generates a list of functions
// (in terms of address ranges) by examining destinations of calls and
// function pointer values. Ideally function entry points are associated
// with symbols so that they can be located again in the code database.
// However calls do addresses without are symbol are handled but the
// function has to be rescheduled on each occasion.
// UsedMap is returned as a bit map of all the locations in the executable
//    image that are used and cannot be overwritten with translated code
// FixupList is a list of locations that need to be fixed up with the
//    addresses of particular functions (for function return links and
//    function pointers)

// we examine the executable image and create counts of entry point usage
// to try and identify individual functions - we also mark which words in
// the image cannot be modified by the translated code
UsedMap = cleared
FixupList = empty
entry_list = address 0 (main entry) and address 8 (SWI handler)
foreach word in the executable image do
   if word is function pointer data and not in filler code then
      // if we have a function pointer then we make sure the addressed
      // function is an entry point - we add the entry word to the fixup list
      // so that a link is written to the start of the function
      mark word as used in UsedMap
      add addressed function to entry_list with infinite usage count
      add addressed function to FixupList (so that link is written to start
         of function)
      if addressed function is marked as being implemented in hardware or
                              is an interrupt function then
         report an error
      endif
   else if word is data and not in filler code then
      // all data is used except if it is in filler code (used for
      // shadow code) where the whole image can be overwritten
      mark word as used in UsedMap
   else if instruction is a SWI then
      // a SWI instruction needs to be presevered (for the immediate value)
      // and the following word is used to hold the link return address
      mark SWI instruction as used in UsedMap
      mark following instruction in FixupList with address of return from
         the SWI handler
   else if instruction is indirect branch for switch code then
      // for switch using an indirect branch we identify all the possible
      // destinations - each is written with a link to the translated
      // implementation of the code
      determine the range of possible values (by analysing range check code)
      foreach switch_dest in the possible range do
         mark switch_dest as used in UsedMap
         mark switch_dest in FixupList so that link to translated code is
            placed in the switch table (if the entry does an immediate direct
            branch then place a shortcut to the destination function)
         mark switch_dest with infinite usage count so that it is a seperate
```

```
            entry point (although may become side entry of a region)
        endfor
    else if instruction is a branch then
        // we increase the reference count of the destination to be able to
        // handle functions that are only entered using a branch
        add destination address to entry_list (or increase reference count
            if already in list)
    else if instruction is a branch and link (i.e. call) then
        // if we have a call then we ensure that the function becomes distinct
        // entry point and arrange for a link return address to be written
        add destination address to entry_list with infinite count
        mark following instruction in FixupList with address of return from
            the function
        if destination function is an interrupt function then
            // destination functions cannot be called directly as they have
            // additional code for preserving registers
            report an error
        endif

    endif
endfor

// we now go through the potential function entry points
func_list = empty
foreach cand_func in entry_list do
    Initially make the function all code between two forced entry points (i.e.
        addressed functions or destinations of calls)
    Revisit all the code within the candidate function and reduce reference
        counts for branch destinations
    if there are non-zero reference counts in the function then
        Subdivide the function between each of the referenced addresses - this
            detects any other side entries that are branched to
    endif
    mark function as critical if entry symbol of function in critical list
    if function not on list of those implemented in hardware and
                            function is not a code filler then
        // do not add behavioural model code for functions implemented in
        // hardware or filler code (to make executable big enough)
        CFLT = CreateCFLT(start address of function, end address of function,
                    func_list (only backward information available))
        add function to func_list along with live inflow information from
            CLFT - this shows which register parameters are used by the function
            (non-volatile registers are masked off from the liveness even through
            they may appear live due to the save and restore code)
    endif
endfor

return func_list
```

13.2.2 Function Scheduling

```
UTTList ScheduleFunction(StartAddr, EndAddr, FuncType, FuncList,
                                    ConnectionReqs)
// Schedules a particular function delimited by given addresses - returns
// a list of the Unit Time Tables (UTTs) for the schedule regions in the
// function
// FuncType is the type of function being scheduled
// FuncList is the full list of functions in the code - this is used
//    to generate better liveness information for the function
// ConnectionReqs are new unit/connection requests used during architectural
//    synthesis

UTT_list = empty
region_list = CreateRegionList(StartAddr, EndAddr, FuncType, FuncList)
foreach CDFG_graph in region_list starting with the deepest do
```

```
    // schedule the CDFG and place the result as shadow code
    UTT = ScheduleCDFG(CDFG_graph, NULL)
    add UTT to UTT_list

    // the function names for monitor code may be listed in the configuration
    // file - such code is only generated for the shadow memory as it is not
    // part of the application
    if the function is not in the monitor code list then
      // Optimise the CDFG
      perform transport optimisation

      // schedule the region
      UTT = ScheduleCDFG(CDFG_graph, ConnectionReqs)
      add UTT to the UTT_list
    endfor

endfor

return UTT_list
```

13.3 Execution Word Optimisation

13.3.1 Functional Unit List Creation

```
FuncUnitList GetFuncUnitList()
// This function creates a list of all the functional units in the
// architecture. This list is used during the placement of
// particular functional units to particular bits in the
// instruction word for control. Each of the functional units will have a
// usage frequency calculated from the first successfully completed schedule
// for each function in the program. These usage frequencies are weighted
// for critical functions and especially the inner loops of critical
// functions to give the those code areas more influence. The width of
// the representation is calculated for each functional unit. This is
// composed of the width for specifying the method plus the widths for
// the selectors for each input operand. The input operand widths may
// grow as more input selections are made available by the addition of
// new connection resources to the processor. A frequency per bit metric
// is calculated for each functional unit. The list of functional units
// is returned.

FU_list = empty
foreach func_unit functional unit in architecture do
  add func_unit to FU_list
  calculate func_unit width from method width, and width of all input
     operands
  calculate frequency_per_bit from frequency of func_unit use / width
endfor

return FU_list
```

13.3.2 Instruction Unit Placement

```
enum LayoutResult {
  Completed,          // indicates the unit was successfully placed in
                      // the instruction word
  MoreOpBitsNeeded,      // indicates that more operation control bits are
                      // required - so the current set of placements
                      // need to be abandoned
  CannotFit           // indicates that there is no position in which to
                      // fit the unit - such a failure indicates that an
                      // instruction representation cannot be created
                      // within the current overall instruction width
}
```

```
LayoutResult PlaceUnit(Unit, SpareBits, FieldList)
// Attempts to place a particular unit in the instruction word. The most
// critical units will have been already placed in individual fields in the
// field list. Ideally a unit is placed into one of the individual fields.
// Alternatively, if the unit representation is wider than the ideal
// existing field then it may be placed into a number of adjacent fields.
// Only adjacent fields are considered so that both the opcode bits for the
// functional unit and the method/select bits for the unit are contiguous.
// A placement is choosen that minimises the total frequency of use for all
// the fields that are included. This minimises the number of occasions on
// which their are clashes between different units that could be activated
// on the same clock cycle.
// Unit is the unit which is to be placed
// SpareBits is the total number of spare bits in the instruction word -
//   this is updated as units are placed
// FieldList is a list of all the instruction fields in the instruction word

candidate_list = empty

// try each field as the base starting field for the unit
for base_field_num = 1 to number of items in FieldList do

   // gradually add adjacent fields to the right hand side until a
   // combined field of sufficient width to represent the unit has been
   // found
   total_freq = frequency of Unit usage
   extra_bits_needed = 0
   num_of_fields = 1
   opcode_width = 0
   available_width = 0
   field_width_enough = false
   for field_num = field_num to number of items in FieldList do

      // the total frequency of the candidate placement is formed from the
      // frequency of the unit being placed plus the frequency of all other
      // units in the field positions it will occupy
      total_freq = total_freq + frequency of all operations currently
                               allocated in field_num field

      // update the total opcode width for the current placement and check if
      // the current field opcode width will need to be extended
      opcode_width = opcode_width + number of selection bits for field
      if all patterns for the opcode bits available for the field_num have
                                   been allocated then
         // all the opcode bit combinations for the field have been used so
         // an extra bit will need to be allocated
         extra_bits_needed = extra_bits_needed + 1
         opcode_width = opcode_width + 1
      endif

      // if the maximum decode width is exceeded then the placement is not
      // valid - each functional unit has a maximum number of bits that it
      // can decode
      if opcode_width > MAXIMUM_DECODE_WIDTH then
         exit loop
      endif

      // increase the available usable width in the placement from all the
      // included fields - exit the loop if it is now wide enough
      available_width = available_width + width of field_num field
      if available_width >= width required for Unit then
         field_wide_enough = true
         exit loop
      endif

      // the number of fields in the placement is increased
      num_of_fields = num_of_fields + 1
```

```
  endfor

  // if we exited because the field was wide enough then this is a
  // candidate placement for the unit - we work out how many extra
  // bits are required above those that are currently spare
  if field_width_enough then
    bit_expansion = 0
    if extra_bits_needed > SpareBits then
      bit_expansion = extra_bits_needed - SpareBits
    endif
    add base_field_num to candidate_list - with attributes of total_freq,
      num_of_fields, extra_bits_needed, bit_expansion
  endif

endfor

// the best candidate placement is choosen
if candidate_list is empty then
  // if there were no candidates then we cannot fit the unit
  return CannotFit
else
  // We find the best candidate - we try and minimise opcode expansion and
  // place the unit with the lowest total frequency. By placing at the
  // lowest total frequency we minimise the chances of clashes between
  // different units. The initial spare bits in the instruction word
  // ensure that the initial placements will not require extra opcode bits
  // (in excess of those that are spare) so placement is made on the basis
  // of minimum frequency.
  find the candidate list with the lowest bit_expansion with secondary
    criteria of lowest total_freq
  if the candidate has a non-zero bit_expansion then
    // if we choose a candidate that requires more bits then the current
    // placements need to be discarded
    return MoreOpBitsNeeded
  else
    // we mark the placement of the unit and reduce the number of spare
    // bits for an internal bit expansion
    place the unit in the required fields - add an extra opcode bit for
      all fields that need it
    SpareBits = SpareBits - extra_bits_needed
    return Completed
  endif
endif
```

13.3.3 Instruction Field Placement

```
FieldList PlaceFields(OpWidth, InstrWidth, FuncUnitList)
// Determines the placement of the instruction fields in the
// instruction word. The instruction fields are allocated on the basis
// of the priority of the input list. As each functional unit
// is allocated to a different functional unit they can be used
// in the same instruction word without interference. Returns the
// list of fields in the instruction word sorted by their frequency.
// OpWidth is the minimum number of bits to be allocated to the opcode
/    fields - this is updated for subsequent field placements
// InstrWidth is the total instruction width
// FuncUnitList is the list of functional units prioritised as required

// work out the number of bits that may be allocated to fields - we
// subtract the number we need to allocate for opcodes and the
// end bit from the total instruction width
bits_left = InstrWidth - OpWidth - 1 (for end bit)

// place the the individual fields in the instruction word in priority order
field_list = empty
foreach unit in FuncUnitList in order do
```

```
   if width of unit < bits_left then
      // If the unit will fit then create a field for it - an extra bit over
      // the actual width of the unit is allocated. This ensures that there
      // will be initial expansion room for opcode bits without having to
      // restart the placement.
      mark unit as being placed
      bits_left = bits_left - (width of unit + 1)
      add unit to field_list
   endif
endfor

// The operation width is updated to include the extra bit in each
// field plus the remainder bits after placing all the fields plus
// one. This the operation width that should be tried for if there
// provide to be insufficient opcode bits and the placement should
// be restarted.
OpWidth = OpWidth + number of entries in field_list + bits_left + 1

// the fields are sorted for placement in frequency order - this ensures
// that if a new unit placement needs to combine a number of fields it
// is able to combine the lower frequency fields at the end of the list
sort field_list so that highest frequency field is first

return field_list
```

13.3.4 Instruction Layout Creation

```
bool GenInstructionLayout(InstrWidth)
// Attempts to place all of the functional units in the instruction word so
// that they can be controlled. Each functional units is controlled by two
// groups of contiguous bits from the instruction word. The control bus
// contains the method and the selectors for all operands. The opcode field
// contains the selector for the functional unit. Each functional unit is
// provided with a fixed selection value which must match the value on the
// opcode bits for the functional unit to be activated. The opcode width may
// vary between 0 and a fixed maximum. Unused bits are set pulled to 0 on
// the input to the functional unit. The algorithm divides the instruction
// word into a number of contiguous fields. The control fields are placed
// adjacently. Opcode bits for each field are also placed adjacently in
// another part of the instruction word. A particular functional unit may be
// controlled by a single field or a group of adjacent fields. Returns false
// if not all functional units could be placed and a wider word is required.
// InstrWidth is the total width of the instruction word

// try different placements with gradually increasing opcode widths -
// we seek to allocate as few bits to the opcode as possible
opcode_width = 0
do
   // obtain a list of the functional units in the architecture
   FU_list = GetFuncUnitList()

   // The functional unit list is ordered on the basis of frequency per bit -
   // thus functional units with the highest use per bit are given the
   // highest priority. Instruction fields are then allocated on that basis.
   // We choose this measure to avoid very wide functional units even if they
   // are frequently used as this would lead to inefficient placement of
   // subsequent units.
   sort FU_list so that highest frequency_per_bit is first
   field_list = PlaceFields(opcode_width, InstrWidth, FU_list)

   // we now try and complete the allocation of the remaining units
   // to fields or combinations of fields
   if field_list is empty then
      // if no fields could be fitted then we need a wider instruction word
      return false
   else
```

```
      sort FU_list so that highest frequency first
      foreach func_unit in FU_list in order do
         if func_unit has not been placed then
            // we try and place a unit
            result = PlaceUnit(Unit, opcode_width, field_list)
            if result = CannotFit then
               // if the unit cannot be fitted anywhere then we need a wider
               // instruction word
               return false
            else if result = MoreOpBitsNeeded then
               // if we need more opcode bits then we need to restart the
               // whole allocation process
               exit loop
            endif
         endif
      endfor

      // return if we managed to place all the units in the instruction word
      if all functional units were placed then
         return true
      endif
   endif

// continue until the opcode width encompases the whole of the usable
// instruction word
while opcode_width < (InstrWidth - 1) repeat

// not all the units could be placed so we need a wider instruction word
return false
```

13.4 Code Scheduling

13.4.1 Region Scheduling

```
UTT ScheduleCDFG(CDFG)
// Schedules a CDFG. Optimisations on the CDFG are undone as
// required in order to generate a schedule. The unoptimised
// CDFG is guranteed to generate a schedule.
// Returns a Unit Time Table (UTT) for the schedule

UTT = empty
perform critical path analysis on CDFG (including weak arcs)
from critical path analysis generate node criticality list

while there are unscheduled nodes do
   Node = the most critical node that can be scheduled (i.e. all
          successors scheduled ignoring weak arcs)
   ScheduleOperation(Node, UTT)
endwhile

// return the created schedule
return UTT
```

13.4.2 Operation Scheduling

```
void ScheduleOperation(Node, UTT)
// Schedule a particular operation in the CDFG - only scheduled if all its
// consumers have already been scheduled
// UTT is Unit Time Table for the schedule

if Node is virtual then
   // Virtual nodes do not have to be actually scheduled - these are
   // conditional nodes for which no conditions were asserted. Examples
```

```
   // are chazs when there have been no order violations or guard
   // instructions which are not required.
   return
else if Node is a register write and earliest dependencies are registers
                         reads of the same register then
   // If we are placing a register write then we can actually merge it with
   // the following read to the same register - no transports need to be
   // placed
   earliest_read = the earliest read operation
   modify earliest_read as being a combined write-read
   return
else if Node is a root (no strong arcs or weak arcs to already
                              issued nodes) then
   // calculate the latest start times for a root node
   weak_last = last cycle in schedule
   strong_last = last cycle in schedule
   if Node has weak successors then
      weak_last = the last cycle taking account of weak arcs with dependents
                         that have not already been issued
   endif
else
   // calculate the latest start times for a standard node
   strong_last = latest theoretical start time for node (subtract arc
         latency from successor times and find earliest ignoring weak arcs)
   weak_last = as strong_last taking account of weak arcs with dependents
                         that have not already been issued
endif

// We need to determine if we are to start searching from the weak or strong
// last time. Ideally we do not want to violate a weak dependency arc so we
// only do so if we can measure and advantage. The difference between the
// weak and strong time must be more than a certain percentage of the depth
// of the tree height to be scheduled.
latest_diff = strong_last - weak_last
tree_height = the height of the tree from Node taking account of latencies
              but ignoring weak arcs
if (latest_diff * fixed percentage) > tree_height then
   // the advantage by violating the weak arcs is more than a certain
   // percentage of the depth of the tree being scheduled - thus we
   // are prepared to violate the weak dependency arcs
   search_start = strong_last
else
   // the advantage by violating the weak arcs is less than a certain
   // percentage of the depth of the tree being scheduled - thus it is
   // not worth violating the weak arc dependencies
   search_start = weak_last
endif

// Place operation at or before the search_time (extend start of schedule if
// required). If it is placed after weak_last then a weak arc dependency
// is violated and and we activate the conditional arc.
for cur_time = search_start to 0 do
   if CanIssue(Node, cur_time, UTT) then
      mark the unit being used by Node as used on the cur_time cycle (and any
         subsequent cycles that it will be blocked)
      if cur_time > weak_last then
         // activate the conditional dependents associated with Node
         foreach weak_arc from Node that is violated do
            mark the Node pointed to by any corresponding conditional arc as
               activated
         endfor
      endif
      return true
   endif
endfor
```

13.4.3 Unit Issueability Check

```
bool CanIssue(Unit, Time, UTT)
// Determines if an instruction to the unit can be issued on the given cycle
// The issue may not be possible depending on interference in the
// instruction layout or if the unit is already being used on the cycle,
// is blocked from a previous usage
// Unit is the particular unit we are trying to place
// Time is the clock cycle for which we are trying the placemend
// UTT is the Unit Time Table for the schedule

if any existing placement on the cycle uses a required instruction field or
                    the unit is blocked from a previous usage then
   return false
else
   return true
endif
```